UNIVERSIDAD DE ZARAGOZA

MÁSTER UNIVERSITARIO EN MODELIZACIÓN E INVESTIGACIÓN MATEMÁTICA, ESTADÍSTICA Y COMPUTACIÓN.

Trabajo de fin de máster:

Inteligencia artificial aplicada a la búsqueda de imagen en el sector eCommerce.

Javier Díez Corral

Directores del trabajo: José Tomás Alcalá Nalvaiz e Inés Aldea Blasco.

Septiembre 2021

Abstract

In the last few years, the great expansion of artificial intelligence combined with progresses made in the deep learning field, have led to the expansions of the techniques and algorithms used in computer vision tasks.

In this work, we will focus in image recognition using semantic segmentation models. This models allow combining classification and detection of the different objects of an image. Particularly, we will analize those models that use fully convolutional neuronal networks (FCN).

We will start with an introduction to semantic segmentation followed by a deep explanation of the elements that conform a FCN distinguishing two phases. The first one will extract the main features of the image and the second one will get back the segmented image.

During this part, we will detail the elements that form the convolutional layers of the network, introducing the convolution operation in this context, the activation functions and the pooling operation used in the first phase and the transposed convolution and the unpooling functions used during the second phase. As an addition to this second part, we will comment some of the most commons architectures of this type of networks.

Once we have defined the elements of this models, we will explain the process of learning carry out by the optimization of a loss function that will allow us to adjust the paremeters of the network in order to minimize the error. In addition, we will introduce the backpropagation algorithm combined with the adaptive moment estimation (Adam) algorithm.

To finish this theoric section, we will define some of the most used semantic segmentation evaluation metrics as well as some other common validation techniques.

In the final part of this notes we will apply some of the theorical concepts introduced in the previous section to study the feasibility of the project on clothing recommendation in an online store of the Efor company.

This project has been developed in Jupyter Notebook and it will be introduced commenting the different steps taken during its development. We will start introducing the data preprocessing phase where we will present an automatic labelling algorithm based on k-means algorithm and on the application of some image processing techniques. Next, we will present the different models that have been adjusted for this project by describing the architecture and the results of each one of them.

In that section, we will mainly focus on the final model that will be finally implemented in our project. This model use a personalized version of the U-Net network. Besides, the categorical cross-entropy loss function and the Adam optimizier will be used for compiling this model. The performance of this model has been quite good, obtaining, in the pixel-wise classification, an accuracy of 94,37 % and 87,04 % in the training and validation set respectively. Besides that, taking into account the clothes classification in the segmented image, the model obtains an accuracy of 90 %.

With the idea of implementing this final model to a real case, we will build a graphical user interface that will emulate a recommender system for an online clothing store. This system will have two recommendation options. The first one will focus on recommendations based on the colours and the type of clothes of the image that is uploaded to the system while the second one will only take into account the type of clothes of the image.

To carry out the first option, we will present an original method using the k-means algorithm combined with the segmentation performed by the model to obtain the most predominant colour of the clothes. For the second option, we will define another criterion that will allow the computer to identify how many different clothes appear in the segmented image.

Finally, we will present the interface of the GUI where we will analyze an example of how both recommendation options work.

Resumen

La gran expansión en los últimos años de la inteligencia artificial combinada con los avances conseguidos en el campo del *deep learning*, han permitido ampliar las técnicas y algoritmos usados en las tareas de visión por ordenador.

En el presente trabajo nos centraremos en la tarea de reconocimiento de imágenes mediante modelos de segmentación semántica. Estos modelos permiten combinar la clasificación y la detección de los diferentes objetos que nos interesen de la imagen. En particular, analizaremos aquellos modelos que utilizan redes neuronales convolucionales completas (FCN).

Comenzaremos realizando una introducción a la segmentación semántica para, a continuación, explicar en profundidad los elementos que componen una FCN distinguiendo dos fases, una encargada de la extracción de características de la imagen y otra encargada de recuperar la imagen segmentada.

A lo largo de esta parte, detallaremos los elementos que forman las capas de convolución de la red, introduciendo la operación de convolución en este ámbito, las funciones de activación y la operación de *pooling* que forman la primera fase y la operación de convolución transpuesta y funciones de *unpooling* utilizadas durante la segunda fase de la red. Como añadido a esta segunda parte comentaremos alguna de las arquitecturas más comunes de este tipo de modelos.

Una vez definidos los elementos que conforman estos modelos, explicaremos cómo se produce el proceso de aprendizaje mediante la optimización de una función de pérdida que permitirá ajustar los parámetros de la red para minimizar el error. En particular, introduciremos el algoritmo de retropropagación combinado con el algoritmo de optimización *Adam*.

Para finalizar esta sección teórica, definiremos algunas de las métricas más utilizadas para validar modelos de segmentación semántica así como algunas otras técnicas de validación más usuales.

En la parte final de esta memoria aplicaremos las herramientas teóricas descritas anteriormente para el estudio de la viabilidad de un proyecto sobre recomendación de prendas de ropa en una tienda online en la empresa Efor.

El proyecto se ha desarrollado en *Jupyter Notebook* y se presentará comentando los diferentes pasos realizados durante su desarrollo. En particular, comenzaremos introduciendo la parte de preprocesamiento de los datos donde presentaremos un algoritmo original de etiquetado de imágenes automático basado en el algoritmo k-medias y en la aplicación de diferentes técnicas de tratamiento de la imagen. A continuación, presentaremos los diferentes modelos que se han ajustado para este proyecto describiendo la arquitectura y los resultados de cada uno de ellos.

Dentro de esta sección, nos centraremos sobre todo en el modelo final que será aquel que implementemos finalmente en nuestro proyecto. Este modelo utiliza una arquitectura que hemos personalizado para este proyecto basada en la red U-Net. Además, para compilar este modelo se utilizará la función de pérdida de entropía cruzada en su versión categórica y el optimizador Adam. Los resultados obtenidos

con este modelo han sido bastante buenos, obteniendo, en la clasificación a nivel de pixel, un 94,37 % y un 87,04 % de precisión en los conjuntos de entrenamiento y validación respectivamente. También cabe destacar la precisión del modelo a la hora de clasificar las prendas de ropa de la imagen siendo del 90 %.

Con la idea de implementar este modelo final a un caso real, se ha construido una pequeña aplicación que emulará a un sistema de recomendación típico de una tienda online. Este sistema poseerá dos opciones de recomendación. La primera tendrá en cuenta, a la hora de recomendar, el color y el tipo de prenda de la imagen que se suba al sistema mientras que la segunda únicamente tendrá en cuenta el tipo de prenda de la imagen.

Para llevar a cabo la primera opción presentaremos un método original utilizando el algoritmo k-medias combinado con la segmentación de la imagen realizada por el modelo para obtener el color más predominante de la prenda de ropa. Además, para llevar a cabo la búsqueda de prendas con el mismo color definiremos un criterio de proporción de similitud entre dos colores. Para la segunda opción, se definirá otro criterio que permitirá identificar a la máquina cuántas prendas diferentes aparecen en la imagen segmentada.

Finalmente, presentaremos la interfaz gráfica de la aplicación dónde se podrán observar, con un ejemplo, las recomendaciones obtenidas al utilizar ambas opciones de recomendación.

Índice general

Capítulo 1

Introducción

1.1. Motivación y objetivos

Hoy en día, con el continuo crecimiento del volumen de información al que los usuarios de Internet tienen acceso, surge la necesidad de crear técnicas o estrategias que permitan resumir o personalizar la información que se le muestra a cada usuario. En particular, vemos cómo cada vez aparecen con más frecuencia en aplicaciones de venta online las opciones de obtener recomendaciones en función de nuestras preferencias. Esta herramienta es conocida como sistema de recomendación, la cual ofrece productos basándose en las preferencias del usuario o en la similitud entre los objetos buscados.

Los sistemas de recomendación, por tanto, incluyen métodos de clasificación, agrupación y asociación. Tradicionalmente, el campo de la minería de datos provee diferentes métodos que permiten realizar las tres tareas anteriores. En particular, como podemos consultar en el capítulo 2 de [24], los primeros y más importantes métodos de clasificación usados para sistemas de recomendación fueron las redes bayesianas, las máquinas de vector soporte y el algoritmo de clusterización k-medias.

Sin embargo, dados los recientes avances en el campo del *deep learning*, [20], y en la inteligencia artificial, han surgido nuevas técnicas en este ámbito que permiten implementar modelos más complejos y que permiten captar las características más abstractas de los datos. De hecho, como se puede consultar en [33], la utilización de estos modelos permite mejorar significativamente los resultados obtenidos por el sistema de reconocimiento debido, principalmente, a la capacidad de incluir partes no lineales en los modelos.

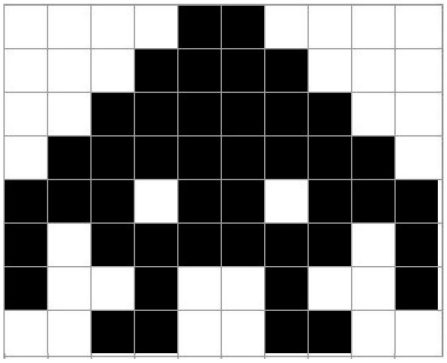
Entre las principales aplicaciones de las técnicas de deep learning, destacan la de visión por ordenador y la de reconocimiento de texto [3]. En este trabajo nos centraremos en la primera de ellas, con el objetivo de construir un sistema de recomendación de prendas de ropa que implemente un modelo que permita la búsqueda por imagen.

Este sistema recomendará prendas de ropa en función de la imagen que suba el usuario. En particular, esta recomendación se hará en base a lo que definiremos más adelante como prendas similares. Este sistema se integrará, además, en una interfaz gráfica de usuario (GUI) de forma que tengamos una herramienta interactiva con la que visualizar los resultados.

Para llevar a cabo esta tarea, necesitamos un algoritmo que sea capaz de procesar y analizar las imágenes, su contenido, los colores que posee, etc. para que puedan ser tratadas por la máquina. De estos procesos se encarga el campo de la visión por ordenador. Antes de comentar el tipo de modelos que usaremos en el trabajo, vamos a explicar cómo entiende el ordenador las imágenes.

Para un ordenador, una imagen es una aplicación $I : U \subset \mathbb{R}^2 \to [0,255]$, donde a cada pixel de

la imagen se le asocia un número que corresponde con su color, en una escala de grises, siendo el 0 el número correspondiente al color negro y 255 el correspondiente al blanco, en el caso de que la imagen sea en blanco y negro. Si es en color, la imagen de la aplicación será un vector de longitud tres, donde cada posición representará la intensidad de los colores rojo, verde y azul respectivamente. Esta codificación se conoce como RGB y permite representar cada color como mezcla de los tres colores primarios. En este caso diremos que la imagen tiene tres canales, uno por cada color primario.



255	255	255	255	0	0	255	255	255	255
255	255	255	0	0	0	0	255	255	255
255	255	0	0	0	0	0	0	255	255
255	0	0	0	0	0	0	0	0	255
0	0	0	255	0	0	255	0	0	0
0	255	0	0	0	0	0	0	255	0
0	255	255	0	255	255	0	255	255	0
255	255	0	0	255	255	0	0	255	255

Figura 1.1: Ejemplo de imagen en blanco y negro con el valor de sus pixeles a la derecha.

Por tanto, el ordenador va a considerar una imagen como una matriz de pixeles. En el caso de una imagen en blanco y negro, tendremos una matriz con dimensiones iguales a las de la imagen $m \times n$ y, si la imagen es en color, tendremos una matriz de dimensión $m \times n$ por cada canal de color RGB. En este último caso diremos que la matriz tiene tres dimensiones, $m \times n \times 3$, correspondientes al ancho, alto y profundo respectivamente. Con esta notación, es claro que una imagen en blanco y negro es aquella que tiene profundidad 1.

Para los problemas de visión por ordenador se utilizan, por lo general, modelos que incluyen redes neuronales convolucionales, [7]. Sin embargo, en este proyecto, hemos decidido abordar este problema de una forma original usando la técnica conocida como segmentación semántica. La principal diferencia entre estas dos técnicas es que mientras la primera solo se centra en la clasificación de los objetos de la imagen, la segunda incluye además la localización de los diferentes objetos. Esta técnica tiene una de sus aplicaciones más recientes en el campo de la conducción automática, [9], donde el coche lleva integrado un dispositivo que le permite distinguir, por ejemplo, los bordes de la carretera, los otros vehículos o los peatones que van por la calzada.

La segmentación de la imagen consiste en dividir una imagen en varias regiones mediante la clasificación de cada uno de sus pixeles en una categoría. Un ejemplo de imagen segmentada se puede encontrar en la figura 2.1. El motivo de elección de la segmentación semántica es el hecho de la combinación de la clasificación y localización de los objetos, lo que nos va a permitir no sólo saber qué tipo de objetos diferentes aparecen en la imagen, sino también extraer información acerca de las características de ellos gracias a que los tendremos localizados dentro de la imagen. Esto nos permitirá construir el sistema de recomendación en base a dichas características.

La información que necesitamos para aplicar este tipo de algoritmos es un poco particular. Aparte de la imágenes originales, necesitamos disponer de unas nuevas imágenes donde cada grupo que pueda aparecer en la imagen original deberá estar pintado de un color diferente en función de la categoría a la que pertenezca. De esta forma, nuestro conjunto de datos estará formado por pares de imágenes compuestos por la imagen original y la imagen pintada. A esta segunda imagen la denominaremos etiqueta y, como veremos a lo largo del trabajo, es la imagen que usa el algoritmo para distinguir las clasificaciones correctas e incorrectas de cada pixel de la imagen.

Para generar estas etiquetas la forma más común de proceder es ir recortando y coloreando manualmente los objetos que aparecen en la imagen. En la parte práctica del proyecto se nombrará alguna de las herramientas que permiten realizar este proceso. Sin embargo, en este trabajo crearemos un nuevo proceso de etiquetado automático y original para este problema basado en el algoritmo de clusterización k-medias y en el uso de diferentes técnicas de tratamiento de la imagen.

1.2. Entorno tecnológico

Otra de las principales razones de la elección del uso de modelos de segmentación semántica sobre otros es la flexibilidad a la hora de elegir el entorno que nos permita implementar este tipo de redes. Los entornos de trabajo elegidos para llevar a cabo el entrenamiento de la red y la creación del sistema de recomendación han sido los siguientes:

- Tensorflow, [29]: Es una biblioteca de código abierto para aprendizaje automático desarrollada por Google y que proporciona una API de Python.
- Keras, [16]: Es una biblioteca diseñada para la experimentación con redes de deep learning escrita en Python. Puede ejecutarse sobre el entorno de Tensorflow.
- Tkinter [31]: Es un paquete de Python considerado un estándar para la interfaz gráfica de usuario y es el que viene por defecto con la instalación de Microsoft Windows.

Para la implementación de estos paquetes y librerías hemos utilizado *Jupyter Notebook*, [15], aplicación web que permite crear cuadernos donde insertar código, material visual y texto narrativo. Esta aplicación tiene un amplio abanico de aplicaciones entre las que se incluyen la simulación numérica, el modelado estadístico y, la que usaremos nosotros, el aprendizaje automático.

Esta aplicación cuenta además con un núcleo encargado de ejecutar el código del cuaderno. Jupyter soporta más de 40 lenguajes de programación como R, Scala o Python que será el que usemos en nuestro caso. Una de las principales razones de la elección de esta aplicación como entorno para ejecutar el código es que permite introducir comentarios en lenguaje LaTeX lo que facilita la introducción de fórmulas matemáticas en celdas adjuntas al código además de la creación de de pequeños informes.

A parte de los entornos de programación, para el desarrollo del trabajo se ha utilizado un portátil cedido por la empresa Efor con un sistema operativo Windows 10 Pro de 64 bits, con un procesador Intel(R) Core(TM) i5-8250U y que posee una memoria RAM de 16GB. Además, destacar que todos los tiempos de ejecución de los modelos realizados durante este trabajo se han realizado con el portátil enchufado a la corriente para intentar obtener el máximo rendimiento.

1.3. Estructura del trabajo

Esta memoria está estructurada en dos grandes capítulos además de este primero que se trata de una breve introducción a la temática del trabajo.

A lo largo del siguiente capítulo se desarrollará toda la teoría relacionada con el tipo de redes utilizadas en el campo de la segmentación semántica. En particular, se explicarán los elementos que componen este tipo de redes y se explicará en profundidad cómo se produce el proceso de aprendizaje de la red así como los métodos más usados para validar este tipo de modelos.

Para finalizar, el último capítulo incluye la puesta en práctica de los procesos descritos en la parte teórica con el objetivo de resolver el problema del sistema de reconocimiento. El capítulo comienza

con la explicación del algoritmo creado para el etiquetado de las imágenes que se van a incluir en el modelo. A continuación, se comentan los resultados obtenidos en los diferentes modelos entrenados y, por último, se presentará el sistema de recomendación programado y la aplicación desarrollada que darán solución al problema en cuestión.

Capítulo 2

Segmentación semántica: Redes Convolucionales Completas

La segmentación semántica es uno de los problemas en el campo de la visión artificial que consiste en dividir una imagen en regiones de acuerdo a las diferentes características que tengan los pixeles de una imagen. De esta forma, el objetivo es el de asignar a cada pixel de la imagen una de las clases del problema. Esta clasificación se realiza teniendo en cuenta las características que comparten los pixeles de la imagen como el color, la intensidad, etc.

Más formalmente, podemos formular el problema de la siguiente forma: dado un conjunto $L = \{l_0 \cdots l_k\}$ de $k+1$ etiquetas o clases, donde l_0, por convenio, es la clase que representa el fondo de la imagen, y un conjunto de variables $X = \{x_1 \cdots x_N\}$, el problema consiste en asignar una etiqueta a cada elemento del conjunto de variables. Por ejemplo, en el caso de una imagen en dos dimensiones, es decir, en blanco y negro, nuestro conjunto X estará formado por los pixeles de dicha imagen. El tamaño de X vendrá dado por el número de pixeles $N = W \cdot H$ donde W y H representan las dimensiones, ancho y alto respectivamente, de la imagen. Para nuestro proyecto, sin embargo, trataremos con imágenes a color por lo que cada pixel de la imagen tendrá tres valores asociados, uno por cada canal de color RGB, de forma que para cada uno de estos canales se tendrá una matriz de dimensión $W \times H$. Por tanto, nuestro conjunto de datos X será una matriz de tres dimensiones $W \times H \times 3$.

(a) Imagen original de [4]. (b) Imagen segmentada.

Figura 2.1: Ejemplo de imagen segmentada donde cada pixel se colorea de acuerdo a la clase a la que pertenece.

Para abordar este problema, una de las técnicas de deep learning más utilizada y que mejores resultados da es el uso de modelos que implementan redes neuronales convolucionales (CNN). Las arquitecturas de estos modelos permiten emular las conexiones entre neuronas que existen en el cerebro humano, encargadas de recibir y transmitir información al resto de neuronas. Los elementos del modelo se organizarán en capas donde se utilizarán operaciones matemáticas para extraer la información del objeto de

entrada a la capa. A los elementos de salida de cada capa los llamaremos mapas de características, cuya información se transmitirá a la siguiente capa.

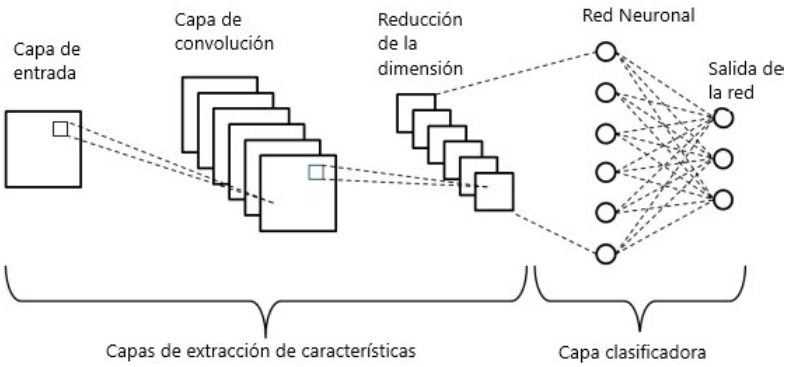


Figura 2.2: Esquema de la arquitectura de una CNN.

Como podemos ver en la figura anterior, la arquitectura de este tipo de redes consiste en una capa de entrada donde se introducen los datos, seguida de una o varias capas de convolución que permiten reducir la dimensión del objeto de entrada y extraer las diferentes características del objeto. Por último, los datos de salida del proceso de convolución se transforman y se introducen en las capas de una red neuronal que es la que finalmente clasificará el objeto.

De esta forma, la información de entrada a la red se va transmitiendo por las distintas capas de la red, con una serie de operaciones intermedias dentro de la capa, para finalmente obtener un número, o una serie de números que permitan clasificar el objeto.

Por tanto, el objetivo de usar un modelo basado en CNN es la clasificación de la imagen de entrada a la red. En segmentación semántica, sin embargo, queremos asignar los pixeles de la imagen de entrada a la clase a la que corresponden por lo que lo ideal sería obtener como salida de la red una imagen segmentada en la que se pueden observar los diferentes elementos clasificados.

Para conseguir este resultado, se usan modelos de CNN en los que solo se usan capas de convolución y se eliminan las capas de la red neuronal encargadas de la clasificación del objeto. Este nuevo modelo se denomina *"fully convolutional network"* (FCN) y obtiene como resultado final una imagen segmentada con los diferentes objetos que hay en ella clasificados. En este trabajo nos centraremos únicamente en este último tipo de modelos. Para encontrar más información sobre CNNs se puede consultar el capítulo 9 de [12].

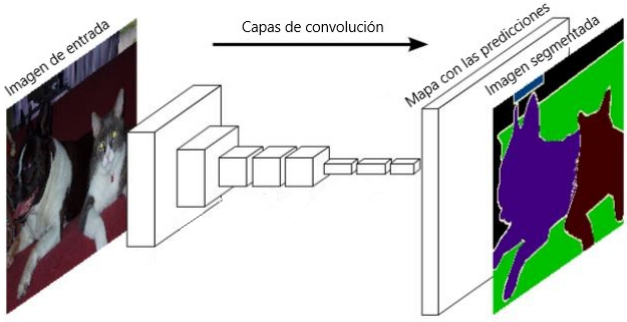


Figura 2.3: Esquema de la arquitectura de una FCN sacada de [9].

En este capítulo nos centraremos en varios aspectos de las redes convolucionales completas. En particular empezaremos con una introducción y explicación de los elementos de la red. A continuación comentaremos alguna de las arquitecturas recientemente utilizadas para el problema de segmentación y, por último, presentaremos las diferentes medidas de precisión y validación usadas en este tipo de redes.

2.1. Elementos de una red convolucional completa

Como ya hemos visto, la principal característica de una red convolucional completa es que todas sus capas son capas de convolución. El funcionamiento de este tipo de redes suele estar dividido en dos fases. La primera, que denominaremos fase de codificación, es la fase de aplicación de sucesivas capas de convolución a la imagen de entrada con el objetivo de extraer las características más relevantes a la vez que se reduce la dimensión de la imagen para afinar este proceso. Al finalizar esta fase, los mapas de características obtenidos tienen poca resolución por lo que para poder ver la segmentación de la imagen correctamente, necesitamos recuperar el tamaño de la imagen de entrada a la red. De este proceso se encarga la segunda fase de la red, que llamaremos fase de decodificación, donde a los mapas obtenidos al terminar la primera fase se le aplican sucesivas capas de deconvolución donde se irán recuperando las dimensiones originales. Al finalizar esta fase, obtendremos un único mapa de características donde cada pixel estará clasificado dentro de una de las clases del problema.

Vamos a comenzar explicando los elementos que forman la fase de codificación de una FCN, introduciendo la operación de convolución, las funciones de activación y la operación de pooling.

2.1.1. Operación de convolución

En general, en el área de análisis la convolución es un operador matemático que aplicado sobre dos funciones f y g de argumento real se obtiene una tercera función que denotaremos como $f * g$. El resultado de esta operación se puede interpretar como el cambio que se produce en g al superponerla con una versión trasladada de f. En una dimensión, la operación se define como:

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x-t)g(t)\mathrm{dt}. \tag{2.1}$$

Notar que esta es la definición continua de la operación de convolución. En nuestro problema, generalmente los datos de entrada a la red serán discretos. Más aún, si pensamos en el caso en el que la entrada a la red es una imagen, la entrada será un array de tres dimensiones por lo que conviene introducir la convolución discreta en varias dimensiones. En nuestro caso, nos bastará definirla para dos dimensiones ya que trataremos cada canal de color de manera independiente. Se define la convolución discreta en dos dimensiones como sigue:

$$(f * g)(x_1, x_2) = \sum_{-\infty}^{\infty}\sum_{-\infty}^{\infty} f(x_1 - t_1, x_2 - t_2)g(t_1, t_2). \tag{2.2}$$

Sin embargo, en el área de tratamiento de la señal se usa una operación parecida a la convolución que se denomina correlación cruzada, introducida en [12]. Con la misma notación de antes se define la correlación cruzada de dos funciones como:

$$(f \star g)(x) = \int_{-\infty}^{\infty} f(x+t)g(t)\mathrm{dt}. \tag{2.3}$$

Notar que esta definición es parecida a la del producto de convolución. De hecho, la relación entre ambas operaciones es la siguiente:

$$(f \star g)(x) = \int_{-\infty}^{\infty} f(x+t)g(t)\mathrm{d}t = \int_{-\infty}^{\infty} f(x-y)g(-y)\mathrm{dy} = \int_{-\infty}^{\infty} f(x-y)h(y)\mathrm{dy} = (f * h)(x),$$

haciendo el cambio $t = -y$ en la primera integral y llamando $h(y) = g(-y)$.

En algunos textos como en [7] a la operación de correlación cruzada se le denomina también convolución por lo que en este trabajo seguiremos este convenio y llamaremos a ambas operaciones convolución puesto que, como en ningún momento aparecerán ambas operaciones juntas, no habrá confusión entre ambas.

De la misma manera podemos definir la correlación cruzada para datos de entrada discretos como:

$$(f \star g)(x_1, x_2) = \sum_{-\infty}^{\infty} \sum_{-\infty}^{\infty} f(x_1 + t_1, x_2 + t_2) g(t_1, t_2). \tag{2.4}$$

Como ya hemos comentado, nuestro input de la red va a ser una imagen. Como vimos, esta imagen se puede ver como una matriz en la que en cada celda está el valor que representa a un pixel de la imagen. El proceso que se realiza en la capa de convolución es el de realizar el producto de convolución entre una submatriz de la matriz de entrada que denotaremos como I y una matriz de dimensión pequeña que llamaremos filtro o Kernel y que denotaremos por K. Para realizar el producto de convolución usaremos la versión discreta de la definición:

$$S_1(i,j) = (I \star K)(i,j) = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} = I(i+n, j+m) \cdot K(n,m),$$

donde la submatriz y el filtro tienen ambas dimensión $N \times M$.

Este filtro es el encargado de capturar la información sobre las diferentes características de los pixeles de la imagen. Realmente, lo que se está realizando es proyectar la matriz del filtro sobre la matriz de entrada. Por tanto, el filtro se irá desplazando horizontalmente sobre la matriz de entrada y, en cada desplazamiento, se realizará la convolución entre la submatriz correspondiente y el filtro. El número de posiciones que se mueve el filtro en cada paso es un número que se puede fijar para cada capa de convolución dependiendo de nuestro conjunto de datos. Este número se denomina *stride* y generalmente se suele fijar a 1.

Veamos ahora un pequeño ejemplo que clarifique este proceso. Consideremos la siguiente matriz y el siguiente filtro:

$$X = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}, \quad K = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}.$$

Vamos a ver como sería el proceso de convolución con un stride = 1.

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & & \\ & & \\ & & \end{pmatrix} \longrightarrow$$

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & -4 & \\ & & \\ & & \end{pmatrix} \longrightarrow$$

...

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & -4 & 4 \\ 2 & 3 & -2 \\ -2 & 4 & -3 \end{pmatrix}$$

Esta última matriz es el resultado de la operación de convolución. Hemos podido observar cómo la matriz de filtro ha ido recorriendo toda la matriz de entrada realizando los productos entre elementos de la misma posición del filtro y de la submatriz correspondiente, para después sumarlos y obtener el número que se coloca en la celda correspondiente de la matriz de salida. Cada uno de estos elementos de la matriz resultante los denominaremos neuronas ya que son los encargados de transmitir la información a la siguiente capa de nuestra red.

Por ejemplo, con la formulación vista anteriormente, el cálculo más detallado del primer elemento de la matriz de salida es:

$$S_1(1,1) = \sum_{n=0}^{2}\sum_{m=0}^{2} = I(1+n,1+m)\cdot K(n,m) = 1\cdot 0+0\cdot(-1)+1\cdot 0+1\cdot(-1)+1\cdot 5+0\cdot(-1)$$

$$+1\cdot 0+1\cdot(-1)+1\cdot 0 = 3.$$

Gracias a esta operación de convolución, la red puede extraer un nuevo mapa de características del objeto de entrada. En nuestro caso, el nuevo mapa sería la matriz obtenida tras el proceso de convolución. Notar también que con esta operación se ha conseguido reducir la dimensión de la matriz de entrada lo cual tiene la ventaja de reducir el coste computacional después de cada capa.

En general, dada una matriz X de dimensión $n \times m$ y una matriz de filtro K de dimensión $n' \times m'$, la matriz resultante tiene dimensión $\frac{n-n'}{stride}+1 \times \frac{m-m'}{stride}+1$.

Por tanto, podemos observar que el stride que fijemos a la hora de realizar la convolución va a influir bastante en la reducción de la dimensión. Si en nuestro ejemplo anterior realizamos la convolución con un stride $= 2$ el proceso quedaría:

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & \\ & \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 4 \\ & \end{pmatrix} \longrightarrow$$

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 4 \\ -2 & \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix} = \begin{pmatrix} 3 & 4 \\ -2 & -3 \end{pmatrix}$$

Observar que en este caso obtenemos una matriz 2×2 por lo que aplicando strides más altos conseguiremos reducir más la dimensión. Sin embargo, el usar un stride muy grande tiene desventajas. La más importante es que puede que la red no logre captar alguna característica importante de nuestro objeto de entrada. Efectivamente, si nos fijamos en el ejemplo anterior vemos como los elementos que están en los bordes de la matriz sólo participan en las operaciones una única vez en el proceso de convolución. Lo mismo nos ocurre, aunque en menor medida, cuando trabajamos con un stride $= 1$.

Una solución, si queremos mantener más información en una capa o simplemente no queremos que nuestra matriz de salida tenga una dimensión baja, es aplicar una técnica llamada *padding*. Esta técnica consiste en ampliar las dimensiones de nuestra matriz de entrada a la capa. Por ejemplo, en el caso que se aplique un padding de dimensión uno a una matriz de dimensión $n \times m$, esta pasaría a tener dimensión $n+2 \times m+2$ y los nuevos huecos que están sin rellenar se completarían con ceros. De esta forma, no se añade nueva información pero la información de los bordes tomará una mayor importancia, como se muestra en la siguiente figura:

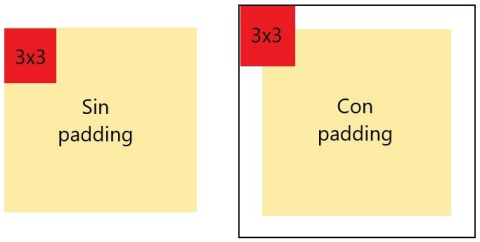


Figura 2.4: Diferencia en el movimiento del kernel dependiendo si no se usa padding o si se realiza con uno de dimensión uno.

Aparte de la elección del stride y de si hacemos uso de la técnica padding o no, podemos también elegir la matriz de filtro que aplicar a nuestro input según las características que nos interese obtener del objeto de entrada. Por ejemplo, en el ejemplo visto anteriormente se ha utilizado un filtro que serviría para enfocar la imagen. Aparte de este, existen otros filtros que permiten detectar los bordes, realzarlos o desenfocar la imagen. Algunos ejemplos de estos tipos de filtros se pueden encontrar en [10].

Para ver un ejemplo visual de la aplicación se puede usar el programa de edición gratuito GIMP, [11], que nos permite introducir manualmente la matriz para convolucionar con la imagen de entrada. Vamos a probarlo con un filtro que nos permite realzar los bordes de una imagen y que viene dado por la matriz:

$$K = \begin{pmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

El resultado de aplicar el filtro es el siguiente:

(a) Imagen original. (b) Imagen filtrada.

Figura 2.5: Aplicación del filtro de realzado de bordes.

Hasta ahora hemos hablado del producto de convolución cuando la matriz y el filtro tienen profundidad 1. Sin embargo, este producto se puede extender cuando tenemos un array tridimensional. En el

caso en el que tengamos una matriz de entrada I de dimensiones $n \times m \times p$ y un filtro K de dimensiones $n' \times m' \times p$, el elemento (i, j) de la matriz resultante viene dado por:

$$S(i,j) = \sum_{l=1}^{p} S_l(i,j),$$

donde los $S_l(i, j)$ son los resultados de la operación de convolución con profundidad uno. Por tanto, notar que cuando se tienen matrices con una profundidad mayor que uno, cada elemento de la matriz resultante S está influido por la suma de los elementos que ocupan la misma posición en las matrices que se obtienen tras las convoluciones parciales.

En resumen, la operación de convolución permite extraer las características o los atributos más relevantes del objeto de entrada. El resultado de la operación son uno o varios mapas de características formados por neuronas. Cada una de estas neuronas estará conectada con las diferentes neuronas de la capa anterior y de la capa siguiente. De esta forma, las neuronas obtenidas en este proceso serán las encargadas de transmitir la información a la siguiente capa de la red.

2.1.2. Funciones de activación

La convolución es una operación lineal por lo que nuestra capa de convolución resultante también es lineal. Sin embargo, muchas imágenes poseen características de naturaleza no lineal por lo que necesitamos una nueva operación que nos permita añadir la no linealidad a esta capa.

Esto último se puede conseguir con la aplicación de una función de activación a cada una de las neuronas obtenidas tras el proceso de convolución. Aparte de añadir la parte no lineal, las funciones de activación permiten decidir si una neurona transmite información a la siguiente capa o no.

De entre todas las funciones de activación, podemos hacer una división entre aquellas utilizadas en las capas internas de la red y aquellas que se utilizan en la capa de salida. De las funciones de activación de las capas internas las más utilizadas actualmente son las funciones ReLU, ELU y la función tangente hiperbólica. Por otro lado, en la capa de salida se suelen utilizar las funciones de activación sigmoide y softmax para problemas de segmentación binarios y multiclase, respectivamente.

En [8] se pueden encontrar otras funciones de activación así como algunas modificaciones de las funciones que acabamos de comentar. Como estas funciones juegan un papel importante en la parte de aprendizaje de la red, definiremos estas funciones y las analizaremos más en profundidad en la sección 2.2.4.

La combinación de la operación de convolución junto con la aplicación de la función de activación a los elementos de salida de la convolución forman una capa de convolución. Tras esta capa, se suele implementar otro tipo de capa cuyo objetivo va a ser reducir la dimensión de los mapas de características obtenidos tras la convolución así como captar la invarianza espacial de los datos. Esta nueva capa se denomina *pooling* y la presentamos a continuación.

2.1.3. Pooling

La etapa de pooling consiste en reemplazar cada elemento del mapa de salida de la capa de convolución por un resumen estadístico de un pequeño entorno de dicho elemento. Estos entornos suelen definirse como rectángulos que van recorriendo la matriz tal y como lo hacía el filtro en la fase de convolución. En cada posición del rectángulo se calcula el resumen estadístico de los elementos que se encuentran dentro de él.

Hay diferentes tipos de pooling. Entre los más usados destaca el llamado *max pooling*, que consiste en devolver el máximo de los elementos del entorno. Otros de los más usados son el *sum pooling*, que devuelve la suma de los elementos del entorno y el *average pooling*, que devuelve la media de los datos del entorno.

La etapa de pooling es bastante útil en técnicas de segmentación semántica puesto que, aparte de reducir la dimensión de los datos, permite capturar la invarianza espacial de las propiedades de los datos de entrada. Es decir, un pequeño cambio que se produzca en los pixeles de la imagen de entrada, no influye en el aprendizaje de la red. Es bastante importante puesto que la salida final de nuestra red tiene que ser la misma imagen de entrada pero segmentada por lo que conviene preservar la información espacial inicial.

Al igual que en la operación de convolución, cuando se aplica el pooling hay que definir el stride que va a marcar el movimiento del entorno a lo largo de la matriz. En general, se suele optar por utilizar un stride$=2$. Por ejemplo, veamos el resultado de aplicar un max pooling de tamaño 2×2 con un stride$=2$ a la siguiente matriz:

$$\left(\begin{array}{cc|cc} 2 & 1 & 3 & 1 \\ 1 & 0 & 1 & 4 \\ \hline 0 & 6 & 9 & 5 \\ 7 & 1 & 4 & 1 \end{array}\right) \longrightarrow \begin{pmatrix} 2 & 4 \\ 7 & 9 \end{pmatrix}. \tag{2.5}$$

De esta parte podemos concluir que de la dimensión de la matriz resultante es $\frac{n-n'}{stride}+1\times\frac{m-m'}{stride}+1$ donde n y m son las dimensiones de la matriz a la que se le aplica el max pooling y n' y m' las dimensiones de los entornos rectangulares.

La sucesión de bloques de capas de convolución y de pooling forman la que habíamos denominado como etapa de codificación. Esta etapa suele finalizar con la aplicación de una o dos capas de convolución, de dimensión 1×1, teniendo la última un número de filtros igual al número de etiquetas de nuestro conjunto de datos. El objetivo de estas últimas capas es el de sustituir a la red neuronal que aparece en las últimas capas de una CNN. En la siguiente figura se puede observar un ejemplo de arquitectura para la fase de codificación sacada de [23].

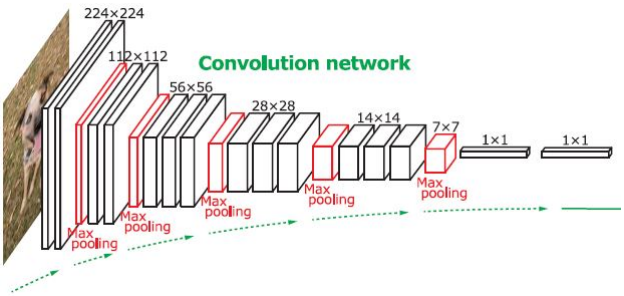


Figura 2.6: Ejemplo de fase de codificacion.

Se puede observar la diferente concatenación de bloques de capas de convolución seguidos de capas de max pooling así como la aplicación de dos capas de convolución con filtros de dimensión 1×1 al final.

Sin embargo, en este punto de la red tenemos varias imágenes de baja resolución en las que se ha perdido información sobre la ubicación de los objetos de la imagen. Es en este momento donde entra en juego la etapa de decodificación de la red.

2.1.4. Etapa de decodificación

Como ya hemos comentado anteriormente por encima, el objetivo en esta etapa va a ser recuperar las dimensiones originales de la imagen. También, en la última capa de esta fase, se utilizará una función de activación que permita devolver la clasificación de los diferentes pixeles de la imagen para poder visualizar la segmentación.

Este proceso puede realizarse utilizando diferentes técnicas que dependerán de la arquitectura de la red que estemos utilizando. En esta sección nos centraremos sobre todo en el método que incluye capas de *unpooling* y de deconvolución.

Como se puede suponer por el nombre, estas capas van a contener las operaciones opuestas a las realizadas durante la fase de codificación. De hecho, esta nueva etapa es una versión simétrica de la realizada anteriormente. Empecemos comentando la etapa de unpooling.

Unpooling

Las capas de unpooling realizan el proceso opuesto al hecho en las capas de pooling. Por tanto, en este caso el resultado de esta operación será un mapa de características de mayor dimensión. Al igual que había diferentes formas de realizar el pooling, también hay diferentes técnicas de unpooling. Para su explicación nos basaremos en los ejemplos introducidos en [26]:

- ***Nearest Neighbor***: Se basa en ir recorriendo los elementos del objeto de entrada y en rellenar las celdas colindantes del objeto de salida con el valor de cada elemento. Por ejemplo, si aplicamos esta técnica a una matriz 2×2 con entornos en la matriz de salida de tamaño 2×2 se obtiene:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \longrightarrow \left(\begin{array}{cc|cc} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ \hline 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{array}\right).$$

  Por tanto, la dimensión de la matriz de salida viene dada en función del tamaño del entorno escogido. En particular, la dimensión de la matriz de salida es $n \cdot n' \times m \cdot m'$ donde n y m son las dimensiones de la matriz de entrada y n' y m' son las dimensiones del entorno escogido.

- ***Bed of Nails***: Es parecido al método anterior lo único que escogemos la celda superior izquierda del entorno de la matriz de salida donde pondremos el elemento correspondiente y el resto de celdas del entorno se rellenan con ceros. Por ejemplo, si aplicamos a la matriz anterior esta técnica con un entorno de tamaño 2×2 obtenemos:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \longrightarrow \left(\begin{array}{cc|cc} 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ \hline 3 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 \end{array}\right).$$

  La dimensión de la matriz de salida sigue la misma fórmula que en la técnica anterior.

- ***Max Unpooling***: Esta técnica aprovecha la simetría que existe entre la fase de codificación y decodificación. Es decir, por cada capa de max unpooling en esta etapa, existe una capa de max pooling en la etapa de codificación.

  El método consiste en guardar las posiciones de las celdas en las que se tiene el máximo elemento del entorno seleccionado en la capa de max pooling que está en la posición simétrica a la de unpooling. Ahora, con esta información colocamos cada elemento de la matriz de entrada de la

capa de unpooling en la celda correspondiente a la posición que habíamos guardado de la capa de pooling. Veamos un ejemplo que va a clarificar esta técnica.

Supongamos que durante la fase de pooling hemos aplicado un max pooling como en el ejemplo de 2.5. Entonces, nos guardamos las posiciones $(1,1)$, $(2,4)$, $(4,1)$ y $(3,3)$. Ahora, continuamos con las etapas de la red y llegamos a la capa de unpooling que está en la posición simétrica a la capa de pooling anterior. Veamos cómo se realiza la operación de max unpooling para la siguiente matriz de entrada:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \longrightarrow \left(\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 \\ \hline 0 & 0 & 4 & 0 \\ 3 & 0 & 0 & 0 \end{array}\right).$$

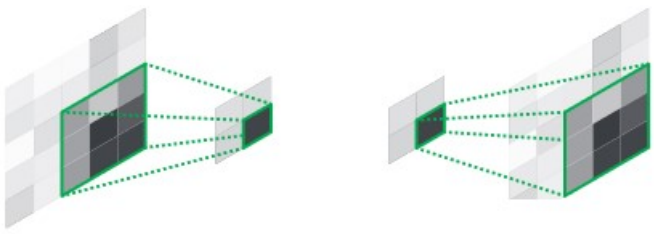
Observamos que cada elemento de la matriz de entrada se ha colocado en la posición que le correspondía en función de las posiciones que nos habíamos guardado de la operación de pooling. Además, la dimensión de la matriz resultante siempre va a ser el doble que la de la matriz de entrada. En este caso, pasamos de una matriz 2×2 a una matriz 4×4.

Esta técnica es la más usada de las tres puesto que, en cierto modo, permite preservar parte de la estructura de la matriz de entrada a la capa de pooling en la fase de codificación.

Notar que en cualquiera de los tres ejemplos, la dimensión de la matriz de salida es el doble que la dimensión de la matriz de entrada. Por tanto, esta capa es bastante útil para aumentar rápidamente la dimensión. Ahora, definamos las capas de deconvolución.

Deconvolución

En las capas de deconvolución se le aplica a cada neurona del objeto de entrada una función de activación y luego al objeto resultante se le aplica la operación conocida como convolución transpuesta. Esta operación se puede ver como una operación opuesta a la realizada en las capas de convolución. De esta forma, vamos a tener los mismos elementos que aparecían en la sección 2.1.1. Tendremos la matriz de entrada, el filtro y un stride que va a marcar el movimiento del filtro. En esa sección anterior, veíamos cómo varias neuronas del objeto de entrada, producían una sola neurona en el objeto de salida. En este caso, con la convolución transpuesta, una neurona del objeto de entrada va a producir varias neuronas en el objeto de salida como se puede ver en la siguiente figura.



(a) Convolución. (b) Convolución transpuesta.

Figura 2.7: Diferencia entre convolución y convolución transpuesta.

Dadas dos matrices A y B, denotaremos la convolución transpuesta como $A**B$. En este caso, no existe una fórmula matemática que permita generalizar este cálculo. Por ello, en este trabajo vamos a presentar dos ejemplos con dos formas diferentes de realizar esta operación. En el primero, vamos a ver como realizar la operación manualmente. Consideremos la siguiente matriz I y el filtro K:

$$I = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}, \quad K = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix}.$$

Poniendo un stride= 1, la convolución transpuesta se realiza como sigue:

$$S = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} ** \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix} = \begin{pmatrix} 0 & 0 & \\ 0 & 0 & \end{pmatrix} + \begin{pmatrix} & 1 & 1 \\ & 2 & 2 \end{pmatrix} + \begin{pmatrix} & & \\ 2 & 2 & \\ 4 & 4 & \end{pmatrix} + \begin{pmatrix} & & \\ & 3 & 3 \\ & 6 & 6 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 \\ 2 & 7 & 5 \\ 4 & 10 & 6 \end{pmatrix}.$$

El resultado es una matriz 3×3. En cada paso de la operación se realiza la multiplicación de un elemento de la matriz de entrada con todos los del kernel y se almacenan en la parte correspondiente de las matrices intermedias. Después, se suman las matrices intermedias obtenidas para obtener la matriz de salida de la capa. Por ejemplo, en el caso anterior, si denotamos como S a la matriz de salida tenemos que:

$$\begin{aligned} S(0,0) &= I(0,0) \cdot K(0,0), \\ S(0,1) &= I(0,0) \cdot K(0,1) + I(0,1) \cdot K(0,0), \\ S(0,2) &= I(0,1) \cdot K(0,1), \\ &\dots \\ S(2,1) &= I(1,0) \cdot K(0,1) + I(1,1) \cdot K(1,0), \\ S(2,2) &= I(1,1) \cdot K(1,1). \end{aligned}$$

Notar que antes de realizar la operación hay que saber la dimensión de la matriz resultante para así poder colocar los elementos intermedios correctamente. En general, si tenemos una matriz $n \times n$ y un filtro de dimensión $m \times m$, la dimensión de la matriz resultante, que denotaremos por s viene dada por la fórmula:

$$s = \text{stride} \cdot (n-1) + m. \tag{2.6}$$

Esta fórmula está sacada de [6] donde se presentan diferentes fórmulas para la dimensión de la matriz resultante según los parámetros que se utilicen. Se puede ver que existe una fórmula más general para la dimensión en caso de que utilicemos padding en la operación de convolución transpuesta que viene dada por la fórmula:

$$s = \text{stride} \cdot (n-1) + m - 2p, \tag{2.7}$$

donde p indica la dimensión del padding.

En el segundo ejemplo vamos a introducir un algoritmo que permite la implementación de este cálculo en lenguaje de código. Vamos a considerar las mismas matrices que en el primer ejemplo. Para desarrollar este ejemplo vamos a usar la idea desarrollada en la sección de convolución transpuesta de [6]. En este artículo se introduce la idea de que la operación de convolución se puede realizar mediante la multiplicación de dos matrices.

Por ejemplo, si una vez obtenida la matriz S quisiéramos convolucionarla con la matriz K podríamos crear una matriz *sparse* donde sus elementos permitan emular el movimiento que haría el filtro a lo largo de S con un stride $= 1$:

$$K' = \begin{pmatrix} 1 & 1 & 0 & 2 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 2 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 2 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 2 & 2 \end{pmatrix}$$

Si además, los elementos de S los ponemos como un vector u de dimensión 9, al hacer la multiplicación de $K' \cdot u$ obtendríamos un vector de dimensión 4 que al convertirlo en una matriz obtendríamos una matriz 2×2 que sería el resultado de la operación de convolución.

Ahora bien, si en vez de considerar la matriz K', consideramos la matriz transpuesta $(K')^T$ y, en vez de multiplicarla por la matriz S, convertida en vector, la multiplicamos por la matriz I convertida en un vector v de dimensión 4, obtendremos el mismo resultado que en el primer ejemplo:

$$(K')^T \cdot v = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 2 & 0 & 1 & 0 \\ 2 & 2 & 1 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 2 \\ 7 \\ 5 \\ 4 \\ 10 \\ 6 \end{pmatrix}$$

Es claro que reorganizando este vector como una matriz 3×3 se obtiene la matriz S. Por tanto, este algoritmo nos permite definir las operaciones de convolución y convolución transpuesta como operación de matrices. Es por esta técnica de cálculo por lo que a la operación se le conoce como convolución transpuesta. Además, este proceso nos permitiría implementar ambas operaciones aunque, para dimensiones grandes, puede conllevar un gran coste computacional debido a la creación de matrices sparse de dimensión muy alta lo que haría que el algoritmo fuese poco eficiente.

Resumiendo, en las capas de deconvolución se realiza el proceso inverso al realizado en las capas de convolución de la etapa de codificación de la red. Primero, se aplica a los datos entrantes una función de activación. A continuación, al resultado de las funciones se le aplica la operación de convolución transpuesta con uno o varios filtros, obteniendo como resultado uno o varios mapas de características que tienen unas dimensiones mayores que las del objeto de entrada a la capa. Estos mapas de salida serán el input de la siguiente capa de unpooling, si existe, o de la siguiente capa de deconvolución en caso de que no exista.

Al igual que en la fase de codificación, estas capas se van sucediendo hasta recuperar las dimensiones de la imagen original. Tras la última capa de deconvolución se aplica una función de activación para poder obtener las probabilidades de pertenencia de cada pixel a cada una de las clases. En la figura 2.8 podemos ver un ejemplo de arquitectura de una red convolucional completa.

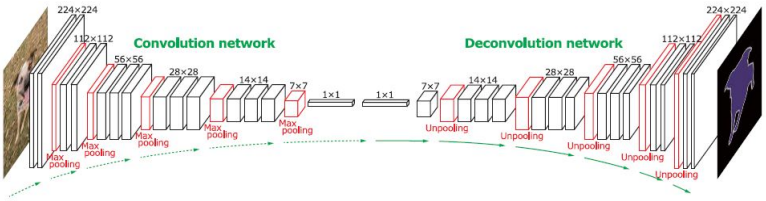


Figura 2.8: Red convolucional completa.

En esta sección hemos comentado la técnica de aumento de dimensión que aparece en [23] donde se presenta la arquitectura de una red conocida como Deconvnet presentada en la figura 2.8. Sin embargo, existen otras técnicas que no requieren del uso de unpooling y permiten perfeccionar el resultado obtenido tras la capa de de deconvolución. Una de las que mejores resultados da es la técnica conocida como *skip connections*.

Esta técnica surge de la necesidad de afinar el resultado obtenido tras la fase de recuperación de la dimensión original de la imagen. La principal idea de este método es guardar los mapas de características obtenidos en las capas iniciales para después concatenarlos con los mapas de características que obtenemos en las capas de la fase de decodificación. Esto permite aprovechar la información de las capas menos profundas, donde se supone que todavía se mantiene parte de la forma original de la imagen, para después combinarla con la información adquirida en las siguientes capas y, de esta forma, poder mantener, por ejemplo, la forma de los bordes de los objetos de la imagen original.

Para ver más clara esta idea, vamos a examinar la arquitectura de la red U-net, presentada en [28], que combina la operación de convolución transpuesta junto con la la técnica de skip connections, como se muestra a continuación:

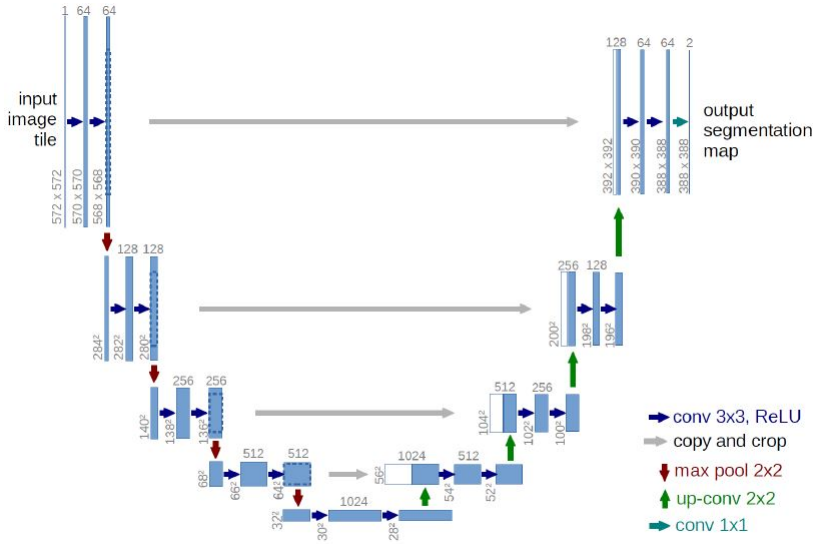


Figura 2.9: Ejemplo de skip connections en la red U-Net.

En esta imagen podemos observar como, tras cada dos capas de convolución en la fase de codificación, se guardan los mapas de características obtenidos. Posteriormente, en la fase de decodificación, tras cada convolución transpuesta se combina el mapa de características obtenido tras esta operación y el mapa de características que se encuentra en la posición simétrica de la red en la fase de codificación. La combinación de estos dos mapas es simplemente una concatenación de sus elementos. Por ejemplo, si tenemos dos mapas de dimensiones $n \times m \times p$ y $n \times m \times q$, el mapa resultante de la concatenación tendría dimensión $n \times m \times (p+q)$.

Tras esta concatenación de los dos mapas, se realizan dos operaciones de convolución para seguir extrayendo características de la imagen. Este proceso se sigue realizando hasta recuperar la dimensión original. En ese momento se aplica al mapa obtenido una capa de convolución con un número de filtros igual al número de clases y de tamaño 1×1 seguido de una función de activación para obtener las probabilidades de pertenencia a una clase de los diferentes pixeles de la imagen.

Para finalizar esta sección vamos a acabar con una reflexión. Hemos observado cómo la técnica que usemos para la fase de redimensión va a depender de la arquitectura de la red. De hecho, no podemos decir que exista una técnica mejor que otra sino que dependerá, entre otras cosas, de nuestro conjunto de datos o del objetivo final del modelo.

En [28], se presentan algunas de las diferentes arquitecturas usadas en segmentación semántica hoy en día como pueden ser las ya comentadas U-Net y Deconvnet u otras como puede ser la red SegNet que

también incluye las fases de codificación y decodificación. Y, en la sección 4 de [9], podemos encontrar otro tipos de técnicas como pueden ser el uso de redes neuronales recurrentes.

2.2. Aprendizaje de la red

Una vez hemos presentado todos los elementos de una red neuronal completa podemos explicar cómo se produce el proceso de aprendizaje de la red. El proceso de entrenamiento o aprendizaje de la red se va a basar en el ajuste de los diferentes parámetros de la red hasta conseguir la precisión deseada. En el caso de una red convolucional completa, los parámetros que se van a ir ajustando en la fase de entrenamiento son los filtros de convolución. A estos parámetros se les denomina pesos de la red. En el caso de un filtro de convolución, cada elemento de la matriz filtro es un peso que además, es independiente del resto. Aparte de los pesos de los filtros de convolución se suelen introducir sesgos en las operaciones de convolución para evitar la propagación de valores nulos a través de las capas. Por tanto, tanto los elementos de los filtros, como los sesgos tras cada operación serán los pesos a ajustar durante la fase de entrenamiento. Este proceso, en general, va a depender de la arquitectura que usemos para nuestra red y del conjunto de datos con el que estemos trabajando.

Antes de comenzar el entrenamiento de nuestra red conviene separar nuestro conjunto de datos en dos subconjuntos llamados conjunto de entrenamiento y conjunto test o de validación. Normalmente, el conjunto de entrenamiento está formado por el 80 % de los datos del conjunto inicial mientras que, el resto de datos, pasan a formar el conjunto de validación. El objetivo de esta división es el de poder evaluar un posible sobreajuste de nuestra red a los datos de entrenamiento, es decir, que nuestra red aprenda muy bien a clasificar los datos con los que se ha entrenado la red, pero, sin embargo, las predicciones en un conjunto de datos distinto sean muy malas. Por tanto, trabajar con estos dos conjuntos nos va a permitir entrenar la red con el conjunto de entrenamiento y, luego poder medir la precisión de clasificación usando los datos del conjunto de validación. De esta forma, se puede ir refinando la arquitectura de la red y quedarnos finalmente con la que mejor resultados proporcione.

El objetivo final del ajuste de este proceso es encontrar el valor de los pesos que permitan minimizar el error en las predicciones. Para poder medir el error cometido durante el proceso de entrenamiento introducimos el concepto de función de pérdida que permite calcular la distancia entre el valor predicho por la red y el valor real de la etiqueta. A esta función de pérdida de la red la denotaremos como $\mathscr{L}$.

2.2.1. Funciones de pérdida

Dependiendo de las características del problema podemos usar diferentes funciones de pérdida. Vamos a suponer que tenemos una imagen de N pixeles y que en nuestro problema hay $k+1$ etiquetas. Denotemos también $Y = \{y_i\}$ e $\hat{Y} = \{\hat{y}_i\}$ los valores reales de las etiquetas y los valores predichos por la red respectivamente. Con esta notación, vamos a introducir algunas de las funciones de pérdida más utilizadas:

- ***Cross Entropy***:

$$\mathscr{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=0}^{k} y_i^c \cdot log\hat{y}_i{}^c,$$

  donde y_i^c es un indicador binario que toma el valor 1 si la etiqueta c está correctamente clasificada para el pixel i e $\hat{y}_i{}^c$ es la correspondiente probabilidad predicha para esa clase.

- ***Focal Loss***:

$$\mathscr{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=0}^{k} (1-\hat{y}_i{}^c)^{\gamma} y_i^c \cdot log\hat{y}_i{}^c,$$

donde γ es un hiperparámetro mayor o igual que cero. Notar que cuando este parámetro es cero, la función de pérdida es la misma que la de *Cross Entropy*. Cuanto mayor sea el parámetro, mayor relevancia tendrán en la función pérdida las clases que tengan poca probabilidad de ser asignadas al pixel correspondiente.

- ***Dice Loss***:

$$\mathscr{L} = 1 - \frac{2\sum_{i=1}^{N}\sum_{c=1}^{C} y_i^c \cdot \hat{y}_i{}^c}{\sum_{i=1}^{N}\sum_{c=1}^{C}(y_i^c)^2 + \sum_{i=1}^{N}\sum_{c=1}^{C}(\hat{y}_i{}^c)^2}.$$

  Esta función permite optimizar el coeficiente de *Dice* que es una de las métricas de validación más utilizadas en segmentación semántica.

- ***IoU Loss***:

$$\mathscr{L} = 1 - \frac{\sum_{i=1}^{N}\sum_{c=1}^{C} y_i^c \cdot \hat{y}_i{}^c}{\sum_{i=1}^{N}\sum_{c=1}^{C}\left(y_i^c + \hat{y}_i{}^c + y_i^c \cdot \hat{y}_i{}^c\right)}.$$

  Similar a la función de pérdida *Dice*, esta función se utiliza para optimizar la métrica de validación IoU.

Estas son las cuatro funciones de pérdida más utilizadas en segmentación semántica. En [21] podemos encontrar más funciones de pérdida clasificadas según sus similitudes.

Para que el aprendizaje de la red sea efectivo la red debe encargarse de encontrar los valores de los pesos que permitan minimizar la función de pérdida. De este proceso se encarga el algoritmo de retropropagación que definimos a continuación.

2.2.2. Algoritmo de retropropagación

El algoritmo de retropropagación o *backpropagation* es una técnica que permite minimizar la función de perdida y actualizar los pesos mediante una estimación de la contribución de cada uno de estos al error de predicción. Como ya vimos, en una red neuronal convolucional completa, la información sobre las diferentes características se va propagando hacia adelante por toda la red, lo que se conoce como red *feedforward*. En este caso, el objetivo es estimar el error de la función de pérdida y transmitir esta información a la primera capa de la red para actualizar los pesos de forma que el error se pueda minimizar en la siguiente iteración, por lo que la información del error se irá transmitiendo hacia atrás. Es en este proceso donde se usa el algoritmo de retroprogación que permite calcular la derivada de la función de pérdida con respecto de los pesos de la red. De esta forma, se puede estimar la contribución de cada peso al valor del error de la red.

Antes de introducir este algoritmo vamos a presentar un esquema de como se transmite la información a lo largo de la red hasta llegar a la predicción que nos ayudará posteriormente en la explicación del algoritmo. Para el esquema vamos a utilizar la siguiente notación basada en [1]:

- l_c: Número de capas en la fase de convolución.
- l_{dc}: Número de capas en la fase de deconvolución.
- $L = l_c + l_{dc}$: Número total de capas de la red.
- f_l: Función de activación en la capa l con $l \in L$.
- fp_l: Función de pooling en la capa l con $l \in \{1, ..., l_c\}$.
- fup_l: Función de unpooling en la capa l con $l \in \{l_c + 1, ..., l_{dc}\}$.

- X: Matriz de entrada a la red.
- SC_k^l: Matriz de salida tras el proceso de convolución con el filtro k y aplicación de la función de activación en la capa l de tamaño $n_{SC} \times m_{SC}$.
- O_k^l: Output tras los procesos de pooling o unpooling de la capa l de tamaño $n_{out} \times m_{out}$.
- UP_k^l: Matriz de salida tras aplicar la función de unpooling en la fase de deconvolución en la capa l.
- $K_{n,m,p}^{l,k}$: k filtros de tamaño $n \times m \times p$ de la capa $l \in L$.
- b_k^l: Sesgo k de la capa $l \in L$.

Con esta notación, podemos construir el esquema de la transmisión de la información a lo largo de la red basándonos en [1] y en [12]:

Algorithm 1 Esquema red feedforward

$SC_k^0 = f_0\left(\sum_{q=1}^{p} X * K_{n,m,q}^{0,k} + b_k^0\right)$.
$O_k^0 = fp_0(SC_k^0)$.
Para $l = 1, ..., l_c$:
    $SC_k^l = f_l\left(\sum_{q=1}^{p} O_k^{l-1} * K_{n,m,q}^{l,k} + b_k^l\right)$.
    $O_k^l = fp_l(SC_k^l)$.
end
Para $l = l_c + 1, ..., l_{dc}$:
    $UP_k^l = fup_l(O_k^{l-1})$.
    $O_k^l = \sum_{q=1}^{p} UP_k^l * K_{n,m,q}^{l,k} + b_k^l$.
end
$O_k^{l_{dc}}$.

Hemos comentado que el algoritmo de retropropagación minimiza la función de pérdida calculando la derivada de dicha función con respecto de los pesos. El cálculo de esta derivada se realiza utilizando la regla de la cadena por lo que su expresión se puede obtener fácilmente utilizando las fórmulas introducidas en el esquema feedforward. En nuestro caso, las expresiones de las derivadas de la función de pérdida con respecto de los filtros y sesgos de la red son las siguientes:

- Respecto de los filtros en la fase de convolución:

$$\nabla_{K_{n,m,p}^{l,k}(u,v)} \mathscr{L} = \frac{\partial \mathscr{L}}{\partial K_{n,m,p}^{l,k}(u,v)} = \sum_{i=0}^{n_{out}-1} \sum_{j=0}^{m_{out}-1} \frac{\partial \mathscr{L}}{\partial O_k^l(i,j)} \cdot \frac{\partial O_k^l(i,j)}{\partial K_{n,m,p}^{l,k}(u,v)} =$$

$$= \sum_{i=0}^{n_{out}-1} \sum_{j=0}^{m_{out}-1} \frac{\partial \mathscr{L}}{\partial O_k^l(i,j)} \sum_{i'=0}^{n_{SC}-1} \sum_{j'=0}^{m_{SC}-1} \frac{\partial O_k^l(i,j)}{\partial SC_k^l(i',j')} \cdot \frac{\partial SC_k^l(i',j')}{\partial A_c(i',j')} \cdot \frac{\partial A_c(i',j')}{\partial K_{n,m,p}^{l,k}(u,v)}.$$

- Respecto de los sesgos en la fase de convolución:

$$\nabla_{b_k^l} \mathscr{L} = \frac{\partial \mathscr{L}}{\partial b_k^l} = \sum_{i=0}^{n_{out}-1} \sum_{j=0}^{m_{out}-1} \frac{\partial \mathscr{L}}{\partial O_k^l(i,j)} \cdot \frac{\partial O_k^l(i,j)}{\partial b_k^l} =$$

$$= \sum_{i=0}^{n_{out}-1} \sum_{j=0}^{m_{out}-1} \frac{\partial \mathscr{L}}{\partial O_k^l(i,j)} \sum_{i'=0}^{n_{SC}-1} \sum_{j'=0}^{m_{SC}-1} \frac{\partial O_k^l(i,j)}{\partial SC_k^l(i',j')} \cdot \frac{\partial SC_k^l(i',j')}{\partial A_c(i',j')} \cdot \frac{\partial A_c(i',j')}{\partial b_k^l},$$

donde $A_c(i,j) = \sum_{q=1}^{p} O_k^{l-1} * K_{n,m,q}^{l,k} + b_k^l$.

- Respecto de los filtros en la fase de deconvolución:

$$\nabla_{K^{l,k}_{n,m,p}(u,v)}\mathscr{L} = \frac{\partial \mathscr{L}}{\partial K^{l,k}_{n,m,p}(u,v)} = \sum_{i=0}^{n_{out}-1}\sum_{j=0}^{m_{out}-1}\frac{\partial \mathscr{L}}{\partial O^l_k(i,j)}\cdot\frac{\partial O^l_k(i,j)}{\partial K^{l,k}_{n,m,p}(u,v)} =$$

$$= \sum_{i=0}^{n_{out}-1}\sum_{j=0}^{m_{out}-1}\frac{\partial \mathscr{L}}{\partial O^l_k(i,j)}\sum_{i'=0}^{n'-1}\sum_{j'=0}^{m'-1}\frac{\partial O^l_k(i,j)}{\partial A_{dc}(i',j')}\cdot\frac{\partial A_{dc}(i',j')}{\partial K^{l,k}_{n,m,p}(u,v)}.$$

- Respecto de los sesgos en la fase de deconvolución:

$$\nabla_{b^l_k}\mathscr{L} = \frac{\partial \mathscr{L}}{\partial b^l_k} = \sum_{i=0}^{n_{out}-1}\sum_{j=0}^{m_{out}-1}\frac{\partial \mathscr{L}}{\partial O^l_k(i,j)}\cdot\frac{\partial O^l_k(i,j)}{\partial b^l_k} =$$

$$= \sum_{i=0}^{n_{out}-1}\sum_{j=0}^{m_{out}-1}\frac{\partial \mathscr{L}}{\partial O^l_k(i,j)}\sum_{i'=0}^{n'-1}\sum_{j'=0}^{m'-1}\frac{\partial O^l_k(i,j)}{\partial A_{dc}(i',j')}\cdot\frac{\partial A_{dc}(i',j')}{\partial b^l_k},$$

donde $A_{dc}(i,j) = \sum_{q=1}^{p} UP^l_k * K^{l,k}_{n,m,q} + b^l_k$.

En general, este tipo de redes suele tener una gran cantidad de parámetros por lo que la búsqueda del mínimo de la función de pérdida calculando las derivadas con este algoritmo tiene un alto coste computacional. Por ello, se han desarrollado diferentes optimizadores que ayudan al algoritmo de retropropagación a alcanzar dicho mínimo. En este trabajo nos centraremos únicamente en el optimizador *Adaptive Moment Estimation*, conocido como Adam. En [25] se puede encontrar un breve resumen de distintos optimizadores para redes neuronales.

2.2.3. Adaptative Moment Estimation

Adam es un algoritmo de optimización, presentado por primera vez en [17], basado en el cálculo del gradiente de primer orden para funciones estocásticas y que utiliza el cálculo de los momentos de primer orden y segundo orden del gradiente de la función a optimizar.

Este algoritmo utiliza la técnica del gradiente descendiente, es decir, es un algoritmo iterativo que, en la búsqueda del mínimo, va actualizando los parámetros de la función en la dirección opuesta al gradiente. Esta técnica solo requiere del cálculo de las derivadas de primer orden de la función respecto de todos los parámetros por lo que, en el caso de que la red tenga pocos parámetros esta técnica no tiene un gran coste computacional. Sin embargo, en general, este no suele ser el caso por lo que, en la práctica, se suele dividir el conjunto de datos en pequeños lotes para que sean procesados por el algoritmo por separado. Es aquí donde entra el concepto de función estocástica, que en nuestro caso será una suma de subfunciones evaluadas en los diferentes lotes formados a partir del conjunto de datos.

Por tanto, tenemos un algoritmo iterativo que procesa la información por lotes por lo que vamos a introducir tres términos que nos permitirán entender como funciona este proceso:

- **Época**: Una época es el número de veces en las cuales el algoritmo procesa todo el conjunto de datos, o, todo el conjunto de entrenamiento en caso de que se haya dividido.
- **Lote**: Un lote, también llamado *batch* es cada una de la submuestras en las que se divide el conjunto de datos o entrenamiento. El tamaño del lote es el número de elementos del conjunto de datos que será procesado por el algoritmo en cada proceso de aprendizaje.
- **Iteración**: Una iteración describe el número de veces que el algoritmo ha procesado un lote y tras la cual se actualizarán los valores de los parámetros.

De esta forma, en caso de que el conjunto de entrenamiento este dividido en varios lotes, cada época del algoritmo puede tener varias iteraciones. En particular, para cada una de las épocas que hayamos fijado se tiene que cumplir:

$$\text{Tamaño de lote} \times \text{número de iteraciones} \geq \text{tamaño del conjunto de entrenamiento.}$$

Hemos comentado que tras cada iteración el algoritmo actualiza los valores de los parámetros. Vamos a ver a continuación cuál es el proceso que sigue el algoritmo Adam para actualizar los pesos de la red. La notación utilizada será la misma que en [17]. Vamos a empezar explicando la notación:

- ε: Constante para garantizar la estabilidad numérica.
- t: Iteración número t.
- θ_t: Vector de parámetros en la iteración t.
- $f_t(\theta)$: Función objetivo estocástica evaluada en θ en la iteración t.
- $g_t = \nabla_\theta f_t(\theta_{t-1})$: Calculo del gradiente de la función objetivo respecto de los parámetros en la iteración t.
- $\hat{m}_t$: Estimación del momento de primer orden del gradiente en la iteración t.
- $\hat{v}_t$: Estimación del momento de segundo orden del gradiente en la iteración t.
- α: Tasa de aprendizaje que indica la magnitud del movimiento en la dirección opuesta de las estimaciones de los momentos de los gradientes.
- $\beta_1, \beta_2 \in [0,1)$: Tasas del decrecimiento exponencial para la estimación de los momentos.

El algoritmo inicializa las estimaciones de los momentos a cero y en cada iteración los actualiza siguiendo las dos siguientes relaciones:

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t, \tag{2.8}$$
$$v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot g_t^2. \tag{2.9}$$

Como estas estimaciones se inicializan a cero, si nos quedásemos con estas estimaciones de los momentos, correríamos el riesgo de que estuviesen sesgadas hacia cero en las primeras iteraciones. Por ello, se propone el siguiente cálculo para corregir este problema:

$$\hat{m}_t = m_t/(1-\beta_1^t), \tag{2.10}$$
$$\hat{v}_t = v_t/(1-\beta_2^t). \tag{2.11}$$

Con esta corrección es claro que se pueda evitar el problema de que los estimadores se queden en valores cercanos a cero. Por último, la actualización de los pesos se computa de la siguiente manera:

$$\theta_t = \theta_{t-1} - \alpha \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \varepsilon). \tag{2.12}$$

La elección de los hiperparámetros del algoritmo va a depender, en general, del conjunto de datos con el que se esté tratando. En cualquier caso, los autores proponen fijar por defecto los siguientes valores: $\alpha = 0{,}001$, $\beta_1 = 0{,}9$, $\beta_2 = 0{,}999$ y $\varepsilon = 10^{-8}$.

Para finalizar, un análisis en profundidad sobre la convergencia de este algoritmo así como algunos resultados de la utilización de este optimizador se pueden encontrar de nuevo en [17].

2.2.4. Funciones de activación en la etapa de aprendizaje

Una de las condiciones que se exige para poder aplicar el algoritmo de retropropagación es que la función de pérdida sea diferenciable. También hemos visto cómo para la retropropagación del error, se hace uso de la regla de la cadena que permite propagar el error hacia atrás en la red. Por tanto, a parte de la diferenciabilidad de la función de pérdida también vamos a tener que fijarnos en la función de activación que usemos en cada capa, con el fin de evitar problemas en el cálculo de los gradientes. Vamos a comentar las ventajas y desventajas de las diferentes funciones de activación presentadas en la sección 2.1.2:

- **Función sigmoide**: Esta función de activación es una función no lineal que devuelve valores en el rango $[0,1]$. La mayor ventaja de esta función es que es fácilmente derivable ya que la derivada depende de la propia función:

$$f(x) = \frac{1}{1+e^{-x}} \longrightarrow f'(x) = f(x) \cdot (1 - f(x)).$$

  Sin embargo, como podemos apreciar en la siguiente gráfica de la derivada de la función, para valores de x que sean grandes o que sean muy negativos, la derivada devuelve valores casi nulos.

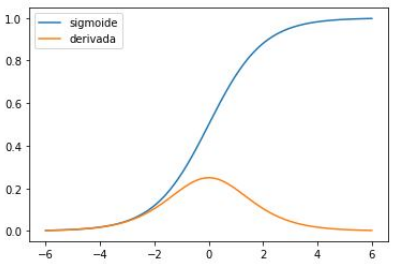


Figura 2.10: Función sigmoide y su derivada.

  Esto produce que, al aplicar el algoritmo de retropropagación con este tipo de funciones, el valor del gradiente sea muy pequeño. Por ejemplo, en una red con n capas, estos valores se multiplicarán n veces lo que producirá que el gradiente decrezca exponencialmente y que la red se entrene más lentamente. De hecho, dependiendo de los valores, puede darse el caso de que la red no aprenda. Este problema es el conocido como *vanishing gradient problem* y provoca, como hemos comentado, que los pesos no cambien sus valores a lo largo de la retropropagación.

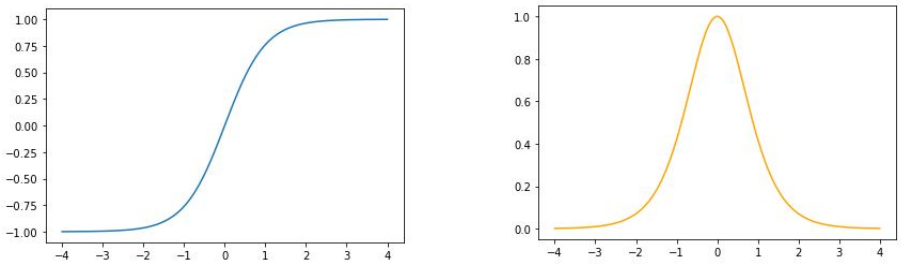
  Por ello, esta función solo se suele utilizar en las capas de salida de la red neuronal para problemas de clasificación binarios.

- **Función tangente hiperbólica**: Como alternativa a la función sigmoide se puede utilizar la función tangente hiperbólica que tiene la ventaja de que es una función con valores centrados en el cero y cuya derivada también depende de la propia función:

$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \longrightarrow f'(x) = 1 - f(x)^2.$$

  Sin embargo, al igual que la función anterior, también sufre del *vanishing gradient problem* debido a que los valores del gradiente son cercanos a cero.

  Una de las ventajas de usar esta función con respecto de la anterior es que la imagen de la función está en el intervalo $[-1,1]$, por lo que para valores que sean muy negativos la función no devuelve el valor cero como ocurría en el caso anterior. Podemos observarlo mirando las gráficas de la función y su derivada:



(a) Función tanh (b) Derivada de la función

Figura 2.11: Función tangente hiperbólica y su derivada.

- **Función ReLU**: Con la necesidad de resolver los problemas de las dos funciones anteriores surge la función ReLU que viene definida de la siguiente forma:

$$f(x) = \text{máx}(0,x) = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{si } x \leq 0. \end{cases}$$

Con esta función, una neurona se activará y transmitirá información a la siguiente capa si tiene un valor positivo. Esta función es una de las más utilizadas debido a que su uso garantiza una optimización de los costes computacionales a la hora de entrenar la red puesto que no incluye términos exponenciales ni divisiones.

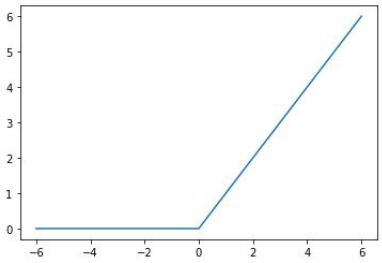


Figura 2.12: Función ReLU.

Notar que esta función es diferenciable en todos los puntos salvo en $x = 0$ lo que puede llevar a pensar que no puede ser usada con algoritmos de optimización basados en el cálculo de gradientes. Sin embargo, para estos casos se define la derivada de la función ReLU de la siguiente forma:

$$f'(x) = \begin{cases} 1 & \text{si } x > 0 \\ 0 & \text{si } x \leq 0. \end{cases}$$

Como podemos observar, para neuronas con valores positivos se evita el problema del *vanishing gradient*. Sin embargo, para neuronas con valores negativos el valor de la derivada es cero lo que causa que algunos gradientes puedan ser nulos y algunas neuronas de la red "mueran" (tengan valor cero). Esto causa que esas neuronas no propaguen información causando que los pesos correspondientes no se actualicen. Este problema es conocido como *dying ReLU problem*.

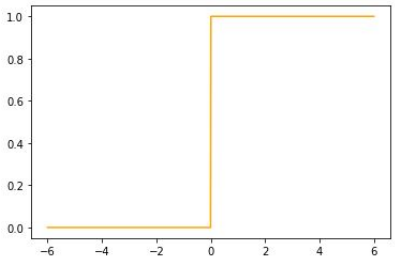


Figura 2.13: Derivada de la función ReLU.

Otro de los problemas de la función ReLU que indican en [8] es que el uso de esta función puede llevar a un sobreajuste del modelo en comparación con otras funciones de activación. De todas formas, dada su simplicidad es la función de activación más usada en las capas ocultas de la red.

- **Función ELU**: La función ELU suele ser utilizada como alternativa a la función ReLU ya que su definición es parecida salvo que devuelve valores distintos de cero para inputs negativos,

$$f(x) = \begin{cases} x & \text{si } x > 0 \\ \alpha e^x - 1 & \text{si } x \leq 0. \end{cases}$$

Esto permite la activación de todas las neuronas distintas de cero lo que soluciona el problema que teníamos con la anterior función. La derivada de la función es la siguiente:

$$f'(x) = \begin{cases} 1 & \text{si } x > 0 \\ f(x) + \alpha & \text{si } x \leq 0. \end{cases}$$

El parámetro α de la función suele ser fijado a 1. Las gráficas de la función y su derivada son las siguientes:

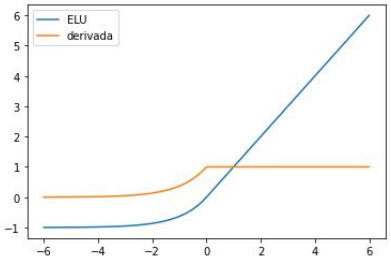


Figura 2.14: Función ELU y su derivada.

Podemos ver también cómo la función de activación ELU puede aliviar el problema del vanishing gradient puesto que salvo para valores muy negativos el gradiente no es nulo.

Una de las limitaciones de esta función es que no está centrada en el cero. Además para valores muy grandes de x produce también valores muy altos lo que puede llevar a una explosión del modelo.

- **Función softmax**: Esta función suele generar pocos problemas en el proceso de aprendizaje de la red ya que es usada en las capas de salida para calcular la distribución de probabilidad de un vector $\mathbf{x} \in \mathbb{R}^n$. Esta función viene dada por:

$$f(\mathbf{x}) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}.$$

Devuelve un valor entre 0 y 1 y suele ser utilizada para problemas multiclase. Por tanto, esta función devuelve un vector de valores que son las probabilidades de pertenencia del objeto de entrada de la red a cada una de las clases del modelo.

2.3. Validación de la red

Una vez hemos visto los elementos que forman una red convolucional completa y cómo se desarrolla el aprendizaje de la red durante el proceso de entrenamiento del modelo, necesitamos definir una serie de métricas que nos permitan medir rigurosamente la actuación del modelo sobre nuevos datos. Para esta tarea se usan los datos del conjunto de validación que habíamos separado previamente del conjunto de datos inicial y que nos van a servir para testear el modelo.

En general, la mayoría de criterios de validación se centran en la medición de la precisión del modelo, aunque, según el proyecto en el que se esté trabajando puede ser más importante centrarse en una rápida ejecución del modelo o en modelos que no consuman mucha memoria a pesar de que la precisión disminuya ligeramente.

2.3.1. Métricas segmentación semántica

Para nuestro propósito, vamos a comentar las métricas más usadas en segmentación semántica e introduciremos una métrica creada específicamente para nuestro modelo. Para ello, seguiremos la notación de [9]. Supondremos que tenemos un total de $k+1$ clases donde la clase 0 será la etiqueta correspondiente al fondo de la imagen. Denotaremos también por p_{ij} a la cantidad de pixeles de la clase i que se han clasificado en la clase j. De esta forma, p_{ii} representa el número de verdaderos positivos y p_{ij} el número de falsos positivos. Con esta notación, presentamos la siguientes métricas de validación:

- ***Pixel Accuracy (PA)***: Es el cociente entre el número de pixeles que se han clasificado correctamente y el número total de ellos,

$$PA = \frac{\sum_{i=0}^{k} p_{ii}}{\sum_{i=0}^{k}\sum_{j=0}^{k} p_{ij}}.$$

  Dentro de esta métrica existe una variante llamada *Mean Pixel Accuracy* donde la tasa de pixeles bien clasificados se calcula en cada clase y luego se calcula la media de los ratios obtenidos,

$$MPA = \frac{1}{k+1}\sum_{i=0}^{k}\frac{p_{ii}}{\sum_{j=0}^{k} p_{ij}}.$$

  Estas son las dos métricas más básicas utilizadas para validar modelos de segmentación semántica.

- ***Intersection over Union (IoU)***: También conocida como índice de *Jaccard*, [30], es un estadístico que permite comparar la similitud de dos muestras A y B de la siguiente forma:

$$J(A,B) = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}.$$

  En nuestro caso, los dos conjuntos serán las imágenes con las etiquetas reales y las imágenes segmentadas por el modelo. De esta forma, la métrica devuelve el valor del cociente entre el número de verdaderos positivos (intersección) y la suma de verdaderos positivos, falsos negativos y falsos positivos (unión) en cada clase. Generalmente, para resumir los valores obtenidos en cada

clase, se suele calcular la media de dichos valores obteniendo una nueva métrica que denotamos como MIoU (Mean Intersection over Union),

$$MIoU = \frac{1}{k+1}\sum_{i=0}^{k}\frac{p_{ii}}{\sum_{j=0}^{k}p_{ij}+\sum_{j=0}^{k}p_{ji}-p_{ii}}.$$

Esta métrica es la más utilizada para tareas de segmentación.

- ***Frequency Weighted Intersection over Union (FWIoU)***: Es una variación de la métrica anterior en la que a cada clase se le asigna un peso en función de la frecuencia de aparición de dicha clase en la imagen,

$$FWIoU = \frac{1}{\sum_{i=0}^{k}\sum_{j=0}^{k}p_{ij}}\sum_{i=0}^{k}\frac{\sum_{j=0}^{k}p_{ij}p_{ii}}{\sum_{j=0}^{k}p_{ij}+\sum_{j=0}^{k}p_{ji}-p_{ii}}.$$

- **Coeficiente *Dice***: Al igual que el índice de Jaccard es un estadístico que permite comparar la similitud de dos conjuntos:

$$Dice = \frac{2|A\cap B|}{|A|+|B|}.$$

Ambos coeficientes se suelen usar indistintamente aunque el valor de este último índice es siempre mayor igual que el de Jaccard.

Por último, hemos definido una métrica especial para nuestro problema. Las métricas anteriores suelen computar los valores en función de los pixeles que han sido bien y mal clasificados en la segmentación. Sin embargo, en nuestro caso, como también nos interesa identificar las diferentes prendas de ropa que aparecen en la imagen, podemos definir una métrica que no tenga tanto en cuenta la localización del objeto en la imagen sino los tipos de prenda de ropa que aparecen en la imagen.

- **Índice de prendas bien clasificadas**: Este índice es un valor entre 0 y 1 que nos indica la cantidad de prendas que han sido bien clasificadas por la segmentación. En particular, de todas las clases que aparecen en la imagen segmentada nos quedaremos con las más representativas (salvo la clase fondo) y compararemos los índices correspondientes a dichas clases con los índices que aparecen en la etiqueta real. Si coinciden, entonces añadimos un 1 a un vector, inicialmente vacío, y, en caso contrario añadimos un 0. Finalmente, sumamos los elementos del vector y dividimos entre su longitud obteniendo un valor en el intervalo $[0,1]$ que será el índice que usemos para validar el modelo.

Notar que todas las métricas de validación anteriores devuelven valores que se encuentran en el intervalo $[0,1]$ por lo que, cuanto más cercano sea el valor a 1, mejor actuará nuestro modelo y, en caso contrario, cuanto más cercano a 0 sean esos valores peor será nuestro modelo.

2.3.2. Otras técnicas de validación

De momento solo hemos comentado algunas medidas que nos permiten validar el modelo de forma analítica. Sin embargo, también podemos hacer uso de elementos gráficos para poder medir la precisión de nuestro modelo. En particular, en esta sección vamos a introducir la matriz de confusión y la curva ROC para problemas multiclase.

Matriz de confusión

La matriz de confusión es una matriz cuadrada de tamaño igual al número de clases que permite comparar la clase real de los pixeles de la etiqueta con la clase predicha por el modelo. En la diagonal de la matriz tendremos el número de verdaderos positivos (TP) de cada clase, y, para cada elemento de la diagonal, los elementos restantes de su fila serán los falsos negativos (FN) y los elementos restantes de su columna serán los falsos positivos (FP). En las celdas restantes tendremos los verdaderos negativos (TN). Un esquema de matriz de confusión con esta forma es el siguiente:

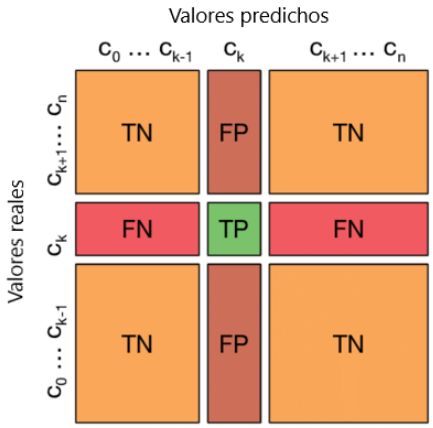


Figura 2.15: Esquema de una matriz de confusión para $n+1$ clases obtenida de [18].

Dado que estamos trabajando con un problema multiclase, a partir de este momento, cuando hablemos de clase positiva nos referiremos a la clase que en ese momento estemos tomando como referencia y clase negativa al resto de clases tratadas como si formasen una única clase.

A partir de esta matriz se derivan nuevas medidas de validación del modelo obtenido. Algunas de las más importantes son:

- **Tasa de bien clasificados (*accuracy*)**: Es la tasa de pixeles bien clasificados respecto del total,

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}.$$

- **Sensibilidad**: Porcentaje de verdaderos positivos respecto de todos los pixeles que son positivos,

$$Sensibilidad = \frac{TP}{TP+FN}.$$

- **Especificidad**: Porcentaje de verdaderos negativos respecto de todos los pixeles negativos,

$$Especificidad = \frac{TN}{FP+TN}.$$

Curva ROC

La curva ROC es un gráfico que permite evaluar la calidad de las predicciones de un clasificador. Generalmente se suele usar para problemas donde se tiene un clasificador binario, aunque en este trabajo extenderemos la interpretación de este gráfico para un problema multiclase. En este gráfico se representa en el eje de abscisas la tasa de falsos positivos frente a la tasa de verdaderos positivos en el eje de ordenadas. Por tanto, se puede interpretar que cuanto más pegada esté la curva a la esquina superior izquierda del gráfico, mayor será la calidad del modelo pues eso significaría que tendríamos una tasa de

verdaderos positivos cercana a uno y una tasa de falsos positivos cercana a cero.

En nuestro caso, donde consideraremos más de dos clases, se obtendrán un número de curvas igual al número de clases del modelo, de forma que la interpretación de cada curva será la misma interpretación que en un problema binario salvo que, en este caso, la clase positiva es la clase de referencia de la curva y, la clase negativa está formada por el resto de clases.

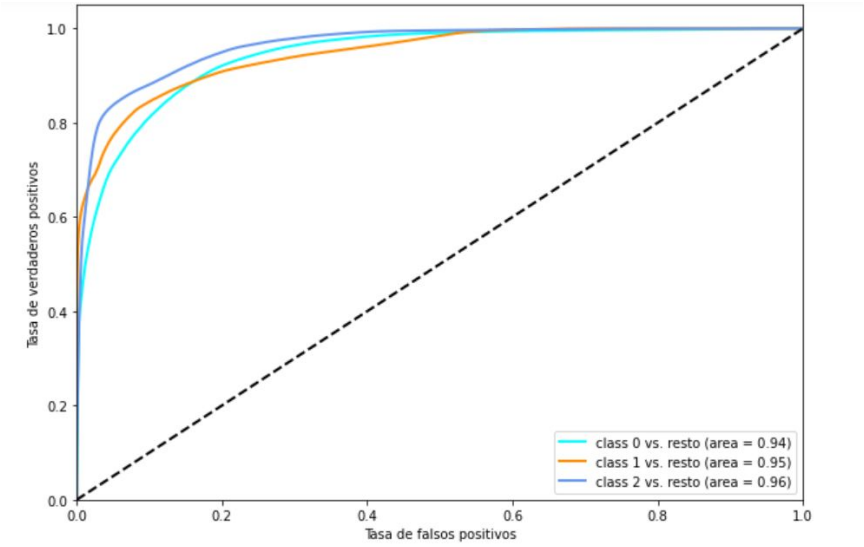


Figura 2.16: Ejemplo de curvas ROC para un problema con tres clases.

En el ejemplo de la figura 2.16 podemos ver cómo en cada curva se compara la clase correspondiente con el resto de clases tratándolas como si fueran una única. La bisectriz del gráfico representa un clasificador cuyas predicciones las clasifica al azar. Por tanto, podemos decir que clasificadores cuya curva ROC esté por encima de la diagonal son mejores que una clasificación por azar y, al contrario, clasificadores con curvas por debajo de la diagonal son peores que un clasificador por azar.

Aparte de para representar las curvas de un modelo, podemos representar en el mismo gráfico las curvas ROC asociadas a diferentes modelos para seleccionar los modelos más óptimos de un conjunto de modelos entrenados.

Por último, del análisis de la curva ROC se puede derivar una nueva médida de validación del modelo que en inglés se conoce como *Area Under de Curve* (AUC). Como su propio nombre indica esta métrica mide el área que se encuentra por debajo de la curva. Para cada una de las curvas tendremos un valor del AUC. Este valor está comprendido entre 0 y 1 y facilita el análisis del desempeño del modelo en cada una de las clases. Generalmente, cuanto más cercano es el valor del AUC a 1, mejor será el modelo. Por ejemplo, en el gráfico anterior los AUCs de las curvas son 0,94, 0,95 y 0,96 para las clases 0, 1 y 2 respectivamente por lo que se podría deducir que ese modelo actúa mejor en la clase 2 cuando se compara con el resto.

La limitación de este análisis es que estamos comparando cada clase con el resto de clases del modelo para hacernos una idea general del modelo por lo que se puede perder información en el caso en el que existan dos clases cuyas predicciones se confundan entre ellas. Por tanto, este gráfico se podría ampliar calculando la curva ROC enfrentando las clases del modelo a pares para conseguir visualizar esta información.

Capítulo 3

Búsqueda de imagen en el sector eCommerce

A lo largo de este capítulo vamos a presentar el estudio y resultados del problema de reconocimiento de imágenes mediante inteligencia artificial propuesto por el área de eCommerce de la empresa Efor. Efor es una empresa con sede principal en Aragón que forma parte del grupo integra dedicada a dar servicios y soluciones tecnológicas para la gestión, comunicación y marketing de las empresas. La solución propuesta en este trabajo va a consistir en llevar a la práctica los aspectos teóricos acerca de segmentación semántica que hemos comentado a lo largo de la memoria.

3.1. Descripción del proyecto

En el mundo de la compra online, las plataformas tradicionales de compras en línea que buscan la información de productos por palabras claves presentan algunos problemas como un gran espacio de búsqueda o la posible inexactitud en los resultados. Desde el sector de eCommerce de la empresa se busca mejorar la experiencia de los clientes implementando un modelo que permita cargar una imagen y obtener productos de aspecto similar mediante la búsqueda por imagen basada en la segmentación. En particular, para este proyecto, las imágenes con las que trabajaremos serán prendas de ropa.

Un proyecto de empresa suele consistir en dos fases diferenciadas. La primera de ellas consiste en un estudio de la viabilidad del problema donde se analiza el conjunto de datos disponible, se seleccionan las diferentes técnicas que se van a utilizar para la resolución y se diseñan los primeros modelos. La segunda es la fase de ejecución donde principalmente se adapta el modelo seleccionado y se realiza un seguimiento y control de su rendimiento en los diferentes campos donde sea implementado.

En nuestro caso, la fase de viabilidad consistirá en el estudio de la aplicación de modelos de segmentación semántica que incluyan el proceso de etiquetado automático mediante el algoritmo k-medias.

Como añadido a la construcción de los modelos se generará una pequeña aplicación que sirva como prototipo o demo para ver el funcionamiento de los algoritmos creados y entrenados. También, como parte de esta aplicación, se construirá una pequeña base de datos donde se recoja la información necesaria para la aplicación de este sistema de recomendación.

Una vez que hayamos visto que tanto las técnicas implementadas como la aplicación son factibles, se pondría en marcha la fase de ejecución. Durante esta fase, se adaptaría la aplicación y el modelo implementado a los requerimientos de la empresa y se realizaría una evaluación de los resultados finales.

Para llevar a cabo este proyecto, contamos con un conjunto de imágenes descargadas de un repositorio abierto el cual se va a introducir y comentar en la siguiente sección.

3.2. Descripción del conjunto de datos

Para este proyecto, vamos a contar con un conjunto de datos formado por 5000 imágenes de prendas de ropa que pertenecen a 20 clases diferentes. La información correspondiente a cada clase viene dada en un fichero excel donde se especifica, entre otras cosas, el nombre de la imagen y su etiqueta asociada. Entre las 20 clases del conjunto de datos, las más comunes son las camisetas de manga corta, los pantalones, los zapatos y los vestidos. Además, el número de imágenes disponibles de cada clase está desbalanceado en el sentido de que por ejemplo, hay cerca de 1000 imágenes de la clase camisetas pero solo 150 de la clase faldas. Además, el conjunto cuenta con dos clases, *others* y *Not sure* donde se encuentran prendas que no pertenecen a ninguna de las 17 clases de ropa que hay en el fichero y prendas que podrían pertenecer a varias a clases del conjunto respectivamente. También se cuenta con una clase minoritaria llamada *Skip* que contiene imágenes que se encuentran documentadas en el archivo excel pero que en la carpeta donde se encuentran las imágenes no existen, bien porque se han eliminado en esta versión, o bien porque no se incluyeron desde un principio. Este conjunto de datos se encuentra en un repositorio abierto y se puede consultar en [13].

Como ya hemos dicho hay bastante diferencia entre el número de imágenes disponibles en cada clase. Para nuestro proyecto vamos a usar 5 clases que hemos elegido según el número de muestras que teníamos en nuestro conjunto y según el interés que pueden tener las prendas de ropa para una tienda online de ropa. En particular, las prendas de ropa elegidas han sido las camisetas de manga corta, los vestidos, los pantalones, los shorts y los zapatos. Como se puede observar, se han elegido prendas de ropa que puedan formar un conjunto entero como camisetas, pantalones y zapatos pero, con el fin de añadir un mayor grado de dificultad al modelo, se ha elegido incluir también los shorts que pueden generar dificultad a la hora de clasificar por su parecido con los pantalones y los vestidos, que tienen una parte superior parecida a una camiseta.

Al ser imágenes que en su mayoría han sido descargadas de la web, contamos con un conjunto bastante heterogéneo en lo que a tipo de imágenes se refiere. Este es uno de los motivos por los que se ha elegido este conjunto de datos ya que nos permite emular el tipo de imágenes que encontramos actualmente en algunas aplicaciones de ropa, como en [32], donde es el usuario el que realiza y sube la foto. En particular, en nuestro conjunto, cada imagen tiene un tamaño diferente, que cambiaremos en la fase de preprocesamiento de los datos, una iluminación y un enfoque diferente y, además, puede haber prendas de ropa que se encuentren cortadas, o que el fondo de la imagen predomine más que la propia prenda, como se ve a continuación:

Figura 3.1: Ejemplos de imágenes con problemas.

Como la segmentación semántica pertenece al campo del aprendizaje supervisado, necesitamos disponer también de las imágenes que contienen las etiquetas de cada pixel. Sin embargo, en nuestro conjunto de datos solo disponemos de las etiquetas globales de la imagen escritas en la hoja de excel. Por tanto, vamos a tener que generar nuestras propias etiquetas para cada imagen de nuestro conjunto de datos. Estas etiquetas van a ser nuevas imágenes donde cada pixel será coloreado de acuerdo a la clase a la que pertenezca. De esta forma, todos los pixeles de una misma clase serán coloreados de la misma forma y formarán lo que denominaremos una máscara de la etiqueta. Por tanto, cada etiqueta tendrá un número de máscaras igual al número de clases que tenga la imagen original. Además, para facilitar la

tarea de coloreado, los pixeles de cada máscara tendrán asignados los mismos valores en los tres canales de color (rgb).

Para realizar el etiquetado de las diferentes imágenes, existen varios programas, [19], [11], que permiten realizar recortes de los elementos existentes dentro de una imagen y, una vez se tienen los diferentes elementos recortados, se asignan las etiquetas correspondientes. La desventaja que tiene este procedimiento es que hay que realizar los recortes a mano para cada imagen lo que llevaría a tener que invertir bastante tiempo en esta tarea. Por ello, en este trabajo proponemos una nueva técnica de etiquetado automática basada en el método de clusterización k-medias.

3.3. Etiquetado con K-medias

3.3.1. Descripción del algoritmo y primer etiquetado

El método de agrupamiento k-medias es un algoritmo no supervisado de clustering. En general, se utiliza cuando queremos hacer una agrupación de los diferentes datos de un conjunto pero no disponemos de las etiquetas que nos permiten identificarlos. El objetivo del algoritmo es el de agrupar las diferentes observaciones del conjunto en k grupos. En nuestro caso, estamos trabajando con imágenes por lo que el objetivo final será agrupar los pixeles de la imagen en k grupos. En particular, como en nuestras imágenes solo se va a tener una prenda de ropa, solo vamos a generar dos etiquetas que serán el fondo y la prenda de ropa correspondiente a cada imagen.

Este algoritmo consta de dos fases dentro de un proceso iterativo. En la primera se calculan los k centroides de los clusters y, en la segunda, se asigna cada pixel al cluster cuyo centroide esté a menor distancia del pixel correspondiente. La definición de la distancia que se use en esta segunda fase va a depender de los datos que estemos utilizando. En general, la distancia más usada es la distancia euclídea.

Una vez que se terminan las dos fases, el algoritmo recalcula los centroides de cada cluster, y, basándose en estos nuevos centros, se vuelven a calcular las distancias entre los pixeles y los centroides de cada cluster reasignándose los pixeles al cluster que esté a menor distancia. De esta forma, el algoritmo continua iterando hasta que en cada cluster la suma de las distancias de los pixeles de dicho cluster al centroide sea mínima. Por tanto, el método de k-medias es un algoritmo que minimiza las distancias entre los pixeles y el centroide correspondiente para cada cluster.

En nuestro caso vamos a tener una imagen de dimensiones $n \times m$ que vamos a clusterizar en dos clusters, uno para el fondo y otro para la prenda de ropa, por lo que k será igual a 2. Usando la misma notación que en [5], denotaremos como $p(n,m)$ los pixeles que van a ser clusterizados y c_k los centroides de cada cluster. Entonces, el algoritmo de k-medias queda:

Algorithm 2 K-medias

1: Inicializa los centros de los clusters aleatoriamente.
2: Para cada pixel de la imagen calcula la distancia euclidea entre el pixel y cada centro:

$$d = \|p(n,m) - c_k\|_2 \tag{3.1}$$

3: Asigna todos los pixeles al cluster más cercano en función de la distancia d.
4: Recalcula la nueva posición de los centros usando la siguiente relación:

$$c_k = \frac{1}{k} \sum_{m \in c_k} \sum_{n \in c_k} p(n,m) \tag{3.2}$$

5: Si se supera el valor fijado de tolerancia, repite los pasos del 2 al 4. En caso contrario, termina.

El algoritmo devuelve los centros de cada cluster y las etiquetas de los clusters a los que ha sido asignado cada pixel. Notar que en el paso 1 la posición de los centros se inicializa aleatoriamente lo que puede llevar a que la clusterización final dependa en exceso de la posición inicial. Para evitar este problema, el lenguaje de programación Python posee una librería, [27], que implementa el algoritmo de k-medias al que se le puede pasar como parámetro el número de ejecuciones del algoritmo que queremos realizar con diferentes semillas. Este parámetro por defecto es 10 y el algoritmo devuelve la mejor clusterización comparando la distancia total que se han movido los clusters durante el algoritmo. La función considera que cuanto menos movimiento haya habido, mejor será la clusterización.

Esta función que hemos comentado va a permitirnos implementar el algoritmo de k-medias en nuestro proyecto. Una de las ventajas que tiene el uso de esta función es que los pixeles de la imagen que tengan intensidades parecidas van a clasificarse en el mismo cluster por lo que, cuanta mayor sea la diferencia en color o intensidad de la prenda de ropa con respecto del fondo, mejor será la clusterización de la imagen. Realmente, este proceso de clusterización lo podemos ver como un proceso de segmentación de la imagen puesto que tendremos los diferentes pixeles de la imagen clasificados en uno de los dos clusters. Esta segmentación de la imagen permitirá construir las máscaras de la etiqueta.

Para medir la calidad de la clusterización realizada por el algoritmo, existen varios coeficientes y métricas de validación. En este trabajo comentaremos únicamente un par de ellos siguiendo la notación de [2]. En ese mismo artículo se pueden encontrar un resumen de algunas de las métricas más utilizadas.

- **Coeficiente de la silueta**: Se define el coeficiente de la silueta como:

$$s = \frac{\sum_{i=1}^{n} S(i)}{n}, \tag{3.3}$$

  donde

  - n es el número de observaciones.
  - $S(i) = \dfrac{b(i) - a(i)}{\max\{a(i), b(i)\}}$.
  - $a(i) = \dfrac{\sum_{j \in \{C_r \setminus i\}} d_{ij}}{n_r - 1}$ es la distancia media de la observación i a todas las observaciones del cluster C_r.
  - $b(i) = \min_{s \neq r}\{d_{iC_s}\}$, donde $d_{iC_s} = \dfrac{\sum_{j \in C_s} d_{ij}}{n_s}$ es la distancia media de la observación i a todas las observaciones del cluster C_s.

  Este coeficiente es un valor dentro del rango $[-1, 1]$ y cuanto más cercano sea a 1 mejor será la agrupación realizada por el algoritmo.

- **Índice de Dunn**: Este índice define la ratio entre la mínima distancia entre clusters y la máxima distancia intracluster que denotaremos como diámetro del cluster:

$$D = \frac{\min_{1 \leq i < j \leq q} d\,(C_i, C_j)}{\max_{1 \leq k \leq q} \operatorname{diam}(C_k)}, \tag{3.4}$$

  donde la distancia entre dos clusters se define como:

$$d(C_i, C_j) = \min_{x \in C_i, y \in C_j} d(x, y) \tag{3.5}$$

  y el diámetro de un cluster como:

$$\operatorname{diam}(C) = \max_{x,y \in C} d(x, y). \tag{3.6}$$

En teoría un buen agrupamiento tendría unos clusters lo más compactos posibles y la distancia entre clusters tendría que ser la máxima posible, por tanto, cuanto mayor sea el valor del índice de Dunn mejor será la clusterización.

Estas medidas las usaremos como criterio para elegir qué imágenes utilizar en nuestro conjunto de datos de entrenamiento. En particular, elegiremos las imágenes que mejores coeficientes obtengan a la hora de usar el algoritmo k-medias para el etiquetado.

Para generar las máscaras, vamos a utilizar la información de las etiquetas asignadas que nos devuelve la función de k-medias. Estas etiquetas las guardaremos en forma de matriz de forma que tengamos identificado el cluster al que pertenece cada pixel. Para generar la etiqueta, el procedimiento será crear una nueva imagen, que será una matriz de ceros del tamaño de la imagen original y, a continuación, asignar a cada celda de esta matriz el número de cluster al que ha sido asignado el elemento que está en la misma posición de la imagen original. Es decir, si por ejemplo el elemento (i, j) de la matriz original ha sido clasificado en el cluster 1, entonces el pixel de la nueva imagen que está en la posición (i, j) tendrá un valor igual a 1 en los tres canales (rgb).

De esta forma, tendremos una nueva imagen formada por dos máscaras, una correspondiente al fondo en la cuál todos sus pixeles tendrán valor 0 y otra correspondiente a la prenda de ropa en la que todos sus pixeles tendrán valor 1. Esta nueva imagen será la etiqueta de la imagen original a la que le hemos aplicado el algoritmo de k-medias.

Para observar el resultado de este proceso, vamos a coger una imagen como ejemplo y vamos a ver como queda su etiqueta. Previamente a aplicar el algoritmo, redimensionaremos la imagen a tamaño 128×128 para que el algoritmo ejecute más rápido.

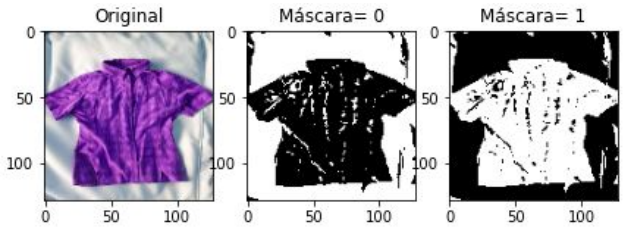


Figura 3.2: Imagen original y máscaras de la imagen etiquetada.

En las imágenes que se encuentran al lado de la imagen original, en la figura 3.2, tenemos las dos máscaras de nuestra imagen etiquetada pintadas de blanco. Los pixeles que tienen valor cero se pintan de blanco en la máscara cero y los pixeles con valor uno se pintan de blanco en la máscara uno. En particular, en la primera podemos observar de color blanco los pixeles que han sido clasificados como fondo y en la siguiente están de color blanco los pixeles que han sido clasificados como prenda de ropa. En general, vemos que la segmentación ha sido buena aunque hay algunos errores de clusterización causados por las arrugas presentes en la camiseta y en el fondo de la imagen original.

Para intentar mejorar este etiquetado y que no sea tan sensible a posibles arrugas o estampados que aparezcan en las prendas de ropa se ha decidido aplicar diferentes filtros a la imagen original antes de ser segmentada así como a la imagen etiquetada tras la clusterización. El objetivo es el de intentar minimizar estos factores que influyen a la hora de etiquetar.

3.3.2. Aplicación de filtros

Filtro a la imagen original

Como ya comentamos en la sección 2.1.1, un filtro era una matriz de baja dimensión que, al convolucionarla con la imagen original, nos daba como resultado una nueva imagen filtrada. En nuestro caso, queremos aplicar filtros que nos permitan eliminar los pequeños detalles de la imagen para que quede lo más lisa posible. Existen varios filtros que permiten distorsionar los elementos de la imagen, como por ejemplo, el filtro que hace borrosa una imagen. Sin embargo, tras realizar varias pruebas con diferentes imágenes del conjunto de datos, los mejores etiquetados que se han obtenido ha sido con la aplicación de una operación de erosión con tres repeticiones, seguido de una operación de dilatación con el siguiente filtro:

$$K = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

Este filtro irá recorriendo la imagen igual que se hacía en la operación de convolución.

La erosión y la dilatación son las operaciones fundamentales de las transformaciones morfológicas en el procesamiento de imágenes. En estas operaciones, el valor de cada pixel de la imagen de salida se basa en una comparación del pixel correspondiente en la imagen de entrada con sus vecinos. La vecindad de cada pixel puede tener diferentes formas viniendo dada por la posición de los unos en el filtro. Por ejemplo, en nuestro caso, cada pixel tendrá una vecindad cuadrangular de tamaño 5×5.

En la operación de erosión, el valor del pixel de salida es el valor mínimo de todos los pixeles de la vecindad lo que nos va a permitir reducir pequeños ruidos o pequeños detalles que tenga la imagen. Además, esta operación encoge ligeramente los objetos de la imagen. Por ello, tras la erosión se suele aplicar la operación de dilatación donde, al contrario que antes, el valor de cada pixel de salida es el valor máximo de todos los pixeles de la vecindad. Esta segunda operación va a evitar que los elementos de la imagen queden muy distorsionados.

Aunque estas operaciones se suelen aplicar a imágenes binarias, las funciones que implementan estos filtros en Python aceptan como argumentos imágenes a color, de forma que cada canal de la imagen es procesado de manera independiente. Veamos, a continuación, cómo cambia nuestra imagen original y la etiqueta tras la aplicación del filtro:

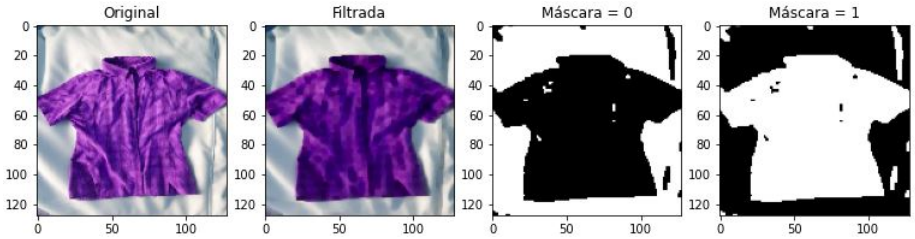


Figura 3.3: Imagen original, imagen filtrada y máscaras de la imagen etiquetada.

Se puede observar cómo en la imagen filtrada se aprecian menos las arrugas y, en general los colores de la camiseta se han oscurecido debido a que la erosión coge el valor mínimo de los pixeles. De esta forma, los colores y la intensidad de los pixeles de la camisa es más homogéneo lo que permite que el algoritmo k-medias pueda clusterizar mejor. De hecho, como se puede observar en las imágenes de las

máscaras, salvo pequeños matices en el fondo y en el interior, el etiquetado se puede considerar bueno.

Filtro a la etiqueta

A pesar de que hayamos conseguido un etiquetado bueno podemos mejorarlo intentando quitar el "ruido" que aparece en la camiseta. En esta imagen no se aprecia mucho "ruido", sin embargo, en prendas de ropa con estampados o de varios colores el etiquetado puede tener más manchas negras en la máscara correspondiente a la ropa. Por tanto, vamos a aplicar un filtro a la etiqueta que contiene las máscaras. Uno de los filtros más usados para eliminar el "ruido" de imágenes binarias es el llamado *median filter*. Este es un filtro no lineal que permite eliminar el "ruido" sin deformar en exceso la imagen.

Al igual que en las operaciones anteriores, el filtro va a examinar cada pixel y su vecindad reemplazando el pixel correspondiente por la mediana de los pixeles que forman el entorno de vecindad. Para este filtro, todos los entornos van a ser cuadrados. Recordar que cada pixel tiene tres valores, uno por cada canal de rgb, por lo que el cálculo de la mediana se realizará de manera independiente para cada uno de los tres canales. Como en nuestro caso estamos tratando con una imagen que, por la construcción que hemos realizado, sus pixeles tienen los mismos valores en los tres canales, explicaremos el proceso haciendo referencia al pixel de la imagen sin distinguir entre los diferentes canales que conforman dicho pixel.

La mediana se calcula ordenando numéricamente de menor a mayor todos los valores de los pixeles del entorno y tomando como pixel de salida el valor que ocupe la posición del medio de esa lista de valores ordenados. En caso de que haya un número par de valores, el pixel de salida será la media de los valores que ocupan las posiciones centrales. Para el caso de los pixeles que se encuentran en los bordes de la imagen, el método replica la información de ese borde tantas veces como sea necesaria para que haya valores dentro de la ventana definida por el entorno. Por ejemplo, para el pixel en la posición superior izquierda de una imagen y un entorno de tamaño 5×5 el proceso sería de la siguiente forma:

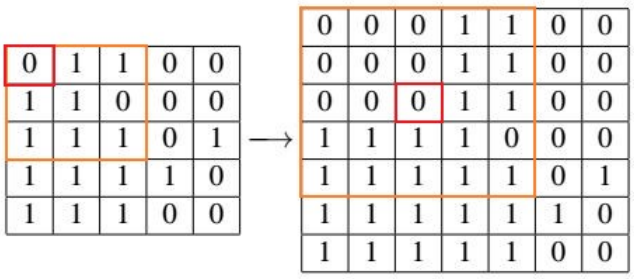


Figura 3.4: Replicado de bordes al usar el *median filter* en los bordes de la imagen.

En este ejemplo podemos observar cómo el pixel de la esquina es cero por lo que es probable que, observando los pixeles de su alrededor, pueda haber sido mal clasificado. Por tanto, al aplicar el filtro, como la mediana del entorno es 1, dicho pixel cambiará su valor.

En nuestro caso, la aplicación de este filtro es bastante sencilla puesto que los pixeles de la imagen que forman la etiqueta solo toman dos valores, cero o uno. Para conseguir un mayor suavizado de la imagen, usaremos entornos de tamaño 7×7. De esta forma, vamos a tener un entorno lo suficientemente grande que nos permita eliminar los pequeños ruidos que puedan surgir tras la clusterización. Veamos ahora, en la figura 3.5, los cambios que se producen al aplicar el median filter al etiquetado obtenido anteriormente.

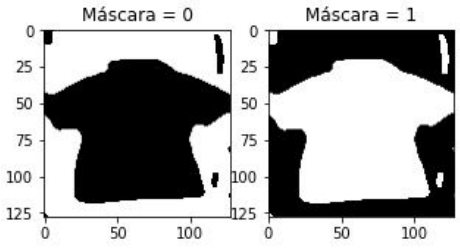


Figura 3.5: Cambio de las etiquetas tras aplicar el median filter.

Comparándolo con el etiquetado anterior, gracias a este filtro hemos conseguido eliminar el ruido del interior de la camiseta así como reducir los pixeles mal clasificados del fondo de la imagen sin deformar en exceso la forma de la prenda de ropa.

En resumen, para mejorar nuestro etiquetado inicial se han aplicado dos filtros. El primero, aplicado a la imagen original, consiste en la aplicación de las operaciones de erosión y dilatación consecutivamente para eliminar los pequeños detalles. Luego, tras el algoritmo de clusterización, el filtro median filter es usado sobre la imagen que contiene las máscaras de la etiqueta con el fin de suavizar la imagen y corregir los pequeños errores que haya podido cometer el algoritmo k-medias a la hora de clusterizar.

3.3.3. Cambio de máscaras

Una de las limitaciones que encontramos al aplicar el algoritmo de k-medias a nuestras imágenes es que las máscaras de la etiqueta obtenida pueden estar intercambiadas. Es decir, el cluster puede haber clasificado los pixeles que forman parte de la prenda de ropa dentro del grupo cero y, los pixeles del fondo, dentro del grupo uno. Por tanto, aunque la agrupación haya sido correcta en el sentido de que los pixeles que forman la prenda de ropa y los pixeles del fondo se encuentran en diferentes grupos, tenemos las máscaras de la etiqueta cambiadas. Podemos ver este problema en la siguiente imagen del etiquetado de una camiseta.

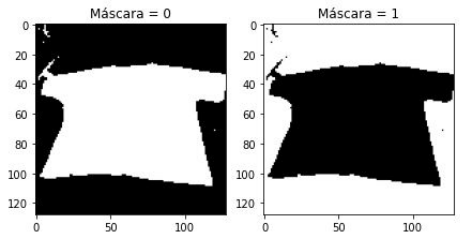


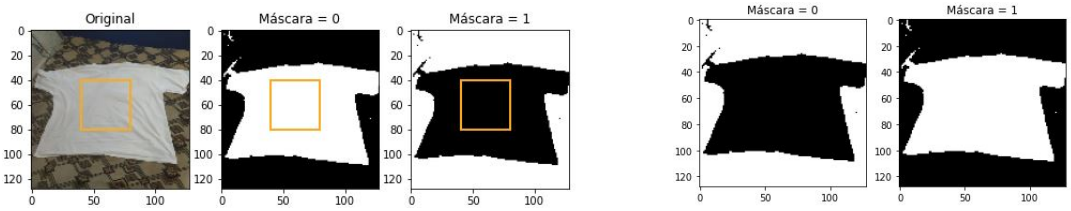
Figura 3.6: Máscaras cambiadas de posición.

Para solucionar este problema, simplemente habría que cambiar el valor de los pixeles de forma que los ceros se conviertan en unos y, al revés, los unos se conviertan en cero. Sin embargo, como queremos que el proceso de etiquetado de las imágenes que vamos a usar para entrenar el modelo esté automatizado, esta tarea no va a ser tan sencilla. Uno de los factores que hay que tener en cuenta es que no todas las etiquetas van a tener las máscaras cambiadas por lo que vamos a tener que determinar un criterio para decidir en qué etiquetas se realiza este cambio.

Inicialmente, se planteó examinar los pixeles que se encontraban en un entorno pequeño de las esquinas de la imagen y si, los pixeles de ese entorno estaban clasificados como fondo (tienen valor cero), no se realizaba el cambio de etiqueta mientras que, en caso contrario, se hacía el cambio. Rápidamente nos dimos cuenta de que esta técnica tenía ciertas limitaciones puesto que, como se puede observar en la figura 3.5, puede haber pixeles de la esquina que se hayan clasificado como prenda de ropa y sin

embargo, no haga falta hacer el cambio pues el etiquetado, en general, ha sido bueno.

Por tanto, la siguiente idea que decidimos llevar a cabo se basó en el hecho de que la mayoría de prendas de ropa se encuentran en el centro de la imagen por lo que, si la etiqueta se ha hecho correctamente debería haber un predominio de pixeles con valor 1 en la parte central. De esta forma, decidimos coger un entorno de tamaño 40×40 entre los valores 40 y 80 de ambos ejes que forman la imagen. Para los pixeles que se encuentran en ese entorno calculamos la media de los valores de dicho entorno y si la media es menor que 0,4, entonces suponemos que esa etiqueta tiene la máscara cambiada y realizamos el cambio. En caso contrario, la etiqueta se queda como está. Para la camiseta anterior con las etiquetas cambiadas se tendría lo siguiente:



(a) Máscaras sin cambiar. (b) Máscaras cambiadas.

Figura 3.7: Vista del entorno definido y cambio de máscaras.

Como se puede observar por el número de máscara, los pixeles que se encuentran dentro del entorno tienen todos valor cero por lo que la media de los valores de ese entorno es cero y se realiza el cambio de máscaras quedando la etiqueta como en el lado derecho de la imagen.

Con esta técnica conseguimos resolver los problemas de las máscaras cambiadas en la mayoría de prendas de ropa. Sin embargo, observamos que para camisetas que tuviesen alguna imagen grande estampada en el centro y para pantalones cuya longitud de pierna era muy larga, cogiendo este entorno central no se obtenían buenos resultados, como se aprecia en la siguiente figura:

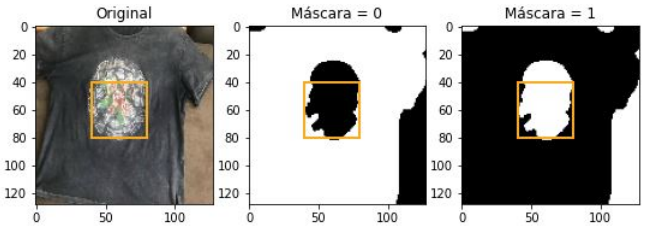


Figura 3.8: Problemas del entorno.

En este caso, el entorno definido cae justo en el estampado de la camiseta que se ha clasificado como fondo por lo que la media del entorno en este caso es de 0,78 de forma que, según nuestro criterio anterior, no se haría cambio de máscaras cuando en realidad sí que habría que realizarlo.

Por tanto, dado que cada prenda de ropa tiene unas características propias decidimos crear un cambio de máscaras personalizado para cada una de las diferentes prendas de ropa que íbamos a usar para entrenar nuestro modelo final. En particular, dividimos las prendas en tres grupos:

- **Pantalones**: Para los pantalones observamos que, en general, donde se agrupan la mayoría de pixeles de la prenda es en la cintura que se encuentra en la parte superior de la imagen por lo que decidimos definir un entorno rectangular entre los valores $[40 : 80]$ en el eje de abcisas y $[20 : 50]$ en el de ordenadas.

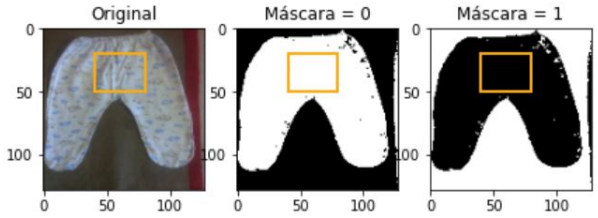


Figura 3.9: Entorno para pantalones.

- **Zapatos**: Para los zapatos observamos que la mayoría de sus pixeles se encontraban en el centro de la imagen, por lo que, para este caso, no cambiamos el entorno de cambio de etiqueta siendo, igual que antes, un cuadrado de tamaño 40×40 entre los valores $[40 : 80]$.

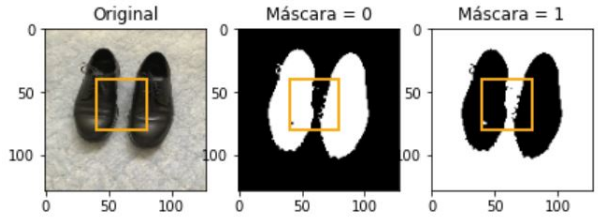


Figura 3.10: Entorno para zapatos.

- **Camisetas, vestidos y shorts**: Dado que estas tres prendas pueden tener estampados o dibujos, al coger únicamente la media de un único entorno corríamos el riesgo de que no hiciese el cambio de etiqueta correctamente como hemos visto anteriormente. Por otro lado, estas tres prendas de ropa suelen tener la mayoría de sus pixeles en el centro de la imagen. Por tanto, lo que hemos hecho ha sido definir un entorno cuadrangular que esté centrado en la imagen, un poco más grande que el utilizado anteriormente y, este entorno dividirlo en 12 cuadrados. De esta forma, para cada nuevo cuadrado se calcula la media de los valores de los pixeles que están en él y a continuación se calcula la media de las medias obtenidas anteriormente. El criterio va a seguir siendo el mismo que anteriormente, si la media final es menor que 0,4, se realiza el cambio de etiqueta.

  Lo que conseguimos con esta partición en cuadrados es minimizar el efecto que tiene sobre la media final un posible dibujo o estampado que tenga la prenda de ropa en el centro de la imagen y que el cluster no haya clasificado como prenda de ropa sino como fondo. Por ejemplo, para la camiseta que antes nos daba problemas quedaría de la siguiente forma:

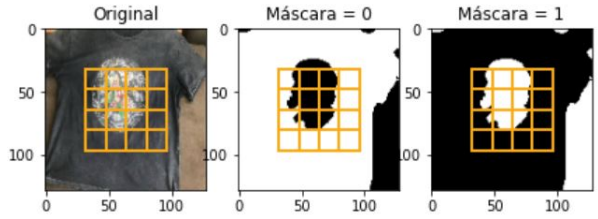


Figura 3.11: Nuevo entorno para la camiseta.

  Como podemos observar, gracias a este nuevo entorno podemos minimizar la influencia que tienen los entornos cuyos pixeles tienen todos valor igual a 1. En particular, con esta nueva técnica, la media obtenida es 0,36 por lo que sí que se realizaría el cambio de máscaras.

Con estos tres entornos definidos para el cambio de máscaras en las etiquetas, tenemos ya un proceso automatizado que nos permite generar la etiqueta de una imagen distinguiendo entre prenda de ropa y

fondo. Sin embargo, dado que el objetivo de nuestro proyecto es realizar también una segmentación de las diferentes prendas de ropa, en el siguiente apartado vamos a generalizar este proceso para el caso en el que tengamos más de dos máscaras en una misma etiqueta.

3.3.4. Etiquetas para el problema multiclase

Como queremos automatizar también este proceso, para generar las diferentes máscaras de la etiqueta en función de la prenda de ropa que haya en la imagen, vamos a ayudarnos del archivo excel que venía junto con las imágenes. También vamos a necesitar de antemano una lista con las diferentes clases que va a haber en nuestro modelo. Además, dado que en nuestro conjunto de datos las imágenes solo tienen una prenda de ropa por imagen, seguiremos aplicando el algoritmo de k-medias con dos clusters. Solo tendremos que preocuparnos de distinguir qué prenda de ropa es la que aparece en la imagen.

El procedimiento que se va a seguir va a ser muy similar al realizado para el caso de dos máscaras, lo único que, en este caso, a los pixeles que el cluster clasifique como cero se les asignará el valor 0 y a los pixeles que el cluster clasifique como uno se les asignará el valor de la posición que ocupa la prenda de ropa de la imagen en la lista de clases de nuestro modelo. Para aclarar este proceso vamos a explicarlo sobre un ejemplo.

Supongamos que tenemos seis clases $L = \{\text{Fondo, Pantalón, Zapatos, Camiseta, Vestido, Shorts}\}$ y que queremos generar la etiqueta de la siguiente imagen:

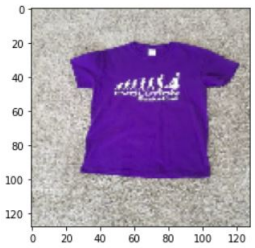


Figura 3.12: Camiseta a etiquetar.

Inicialmente, buscamos en la hoja excel el nombre de la imagen que se quiere etiquetar y guardamos la clase a la que pertenece, en este caso a la clase camiseta. A continuación, se aplican los filtros de erosión y dilatación a la imagen original y se utiliza el algoritmo de clusterización de k-medias. En este momento, ya tenemos los pixeles clasificados por el cluster. Para generar la etiqueta, creamos una nueva imagen donde los pixeles de la clase cero del cluster tendrán valor cero. A los pixeles que están en el grupo 1 se les asigna la posición, empezando a contar desde cero, que tiene la clase camiseta en la lista de clases y que, en este caso, es tres. Por último, aplicamos el median filter y, en caso de que fuera necesario, realizamos el cambio de máscaras cambiando los ceros por treses y viceversa.

Con esto ya tenemos la etiqueta generada y, aunque parezca que solo hemos generado dos máscaras, a la hora de visualizar las máscaras por pantalla, veremos que hay tantas como número de clases, con la particularidad de que cuatro de ellas se verán completamente en negro ya que ningún pixel pertenece a ninguna de esas clases, como vemos en la siguiente figura.

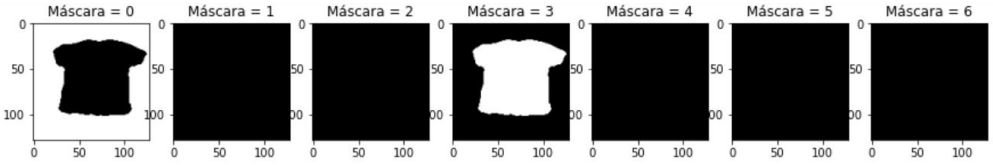
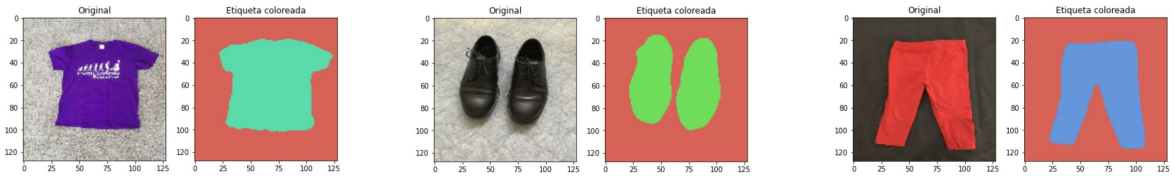


Figura 3.13: Máscaras de la etiqueta de la camiseta.

Otro de los beneficios de asignar a los pixeles la posición de la etiqueta en la lista de clases es que se pueden visualizar las etiquetas de las imágenes coloreadas en función del valor que tengan los pixeles de la etiqueta gracias a una función que hemos programado para este cometido. Por ejemplo, para un pantalón, unos zapatos y la camiseta anterior las etiquetas serían las de la siguiente figura.



(a) Etiqueta de la camiseta. (b) Etiqueta del zapato. (c) Etiqueta del pantalón.

Figura 3.14: Originales y etiquetas coloreadas según la máscara de tres prendas de ropa.

Podemos ver cómo, de esta forma, a la máscara correspondiente al fondo se le asigna el color rojo en las tres etiquetas y, para cada prenda, tenemos colores diferentes en función de la clase a la que pertenezcan.

Resumiendo, hemos conseguido crear un proceso automatizado que nos va a permitir etiquetar las imágenes que usemos para entrenar nuestro modelo. Para compactar toda la información de esta sección a continuación, se presenta cómo quedaría la estructura del algoritmo de etiquetado.

Algorithm 3 Etiquetado

1: Carga la imagen y redimensionala a tamaño 128×128.
2: Busca en la hoja excel la clase a la que pertenece la prenda de ropa de la imagen.
3: Aplica un filtro de erosión 3 veces seguido de uno de dilatación con un filtro de tamaño 5×5.
4: A la imagen filtrada aplícale el algoritmo de k-medias con $k = 2$.
5: Crea una nueva imagen cuyos pixeles tendrán valor 0 si el pixel que ocupa la misma posición en la imagen filtrada pertenece al cluster cero. En caso contrario el pixel tendrá el valor de la posición que ocupe la clase de la prenda de ropa en la lista de etiquetas del modelo.
6: Aplica el median filter a esta nueva imagen.
7: Comprueba si es necesario realizar el cambio de máscaras en esta nueva imagen haciendo uso del entorno correspondiente según el tipo de prenda de ropa. En caso afirmativo, realiza el cambio.
8: Guarda esta última imagen en la carpeta donde estén el resto de etiquetas.

3.4. Creación del modelo

Una vez hemos fijado el proceso que vamos a seguir para realizar las etiquetas de las imágenes que usaremos para entrenar y validar el modelo, ya se puede pensar en la arquitectura que va a tener la red de nuestro modelo. Previamente a presentar la arquitectura, hay que definir el conjunto de datos final así como los conjuntos de entrenamiento y validación que se van a utilizar para entrenar el modelo.

Como ya comentamos, para nuestro modelo vamos a utilizar cinco prendas de ropa. En nuestro conjunto de datos el número de muestras de prendas de ropa es muy distinto siendo los shorts la prenda de la que menos muestras disponemos con 308. Además, a pesar de que los resultados del etiquetado de las imágenes son en general buenos, hay imágenes donde debido a los contrastes de luz o a la mala calidad de la imagen, en el etiquetado no se distinguen bien las formas de la prenda de ropa. Por tanto, a la hora de generar nuestro conjunto de datos final, vamos a seguir dos criterios.

El primero de ellos va a ser la elección de prendas de ropa que permitan formar un conjunto lo suficientemente heterogéneo y que permitan construir un modelo que generalice bien. Por otro lado, vamos a elegir imágenes cuyas etiquetas sean lo suficientemente buenas, es decir, que por lo menos haya una distinción grande entre fondo y prenda de ropa. Este último criterio es necesario, pues si realizamos el entrenamiento con imágenes que tienen etiquetas defectuosas, la red no aprenderá adecuadamente.

La decisión de si el etiquetado es bueno o no, puede tener una componente subjetiva y podemos caer en el error de elegir prendas tan buenas que el modelo final no sea capaz de generalizar. Por ello, para elegir las etiquetas buenas, vamos a basarnos en una de las métricas de validación del cluster k-medias. En particular, usaremos el coeficiente de la silueta que hemos definido en la ecuación 3.3. Tras varias pruebas, se ha determinado que consideraremos una etiqueta buena aquella cuyo coeficiente de la silueta sea mayor que 0,7.

Por tanto, para elegir las prendas que van a formar nuestro conjunto de datos final cogeremos todas las prendas que tenemos de las 5 clases y, eligiendo aleatoriamente, tomaremos una imagen y le aplicaremos el algoritmo de etiquetado. Si el coeficiente de la silueta de la imagen clusterizada es mayor o igual que 0,7, entonces guardaremos dicha imagen en el conjunto de datos final. Este proceso se va a repetir hasta que tengamos 200 imágenes de cada una de las clases. Por tanto, nuestro conjunto final estará formado por un total de 1000 imágenes en las cuales cada clase está representada por 200 imágenes.

Por último, para entrenar el modelo, tanto las imágenes originales como las etiquetadas serán redimensionadas a tamaño 128×128 y, las etiquetas de las imágenes serán transformadas a un formato *one hot encoding*. Este formato transforma la matriz de la imagen de la etiqueta, que tiene profundidad 3, en una con profundidad igual al número de clases y que tendrá en cada celda un vector binario con un 1 en la posición correspondiente a la clase a la que pertenece el pixel de esa celda y cero en el resto de posiciones. Tras este proceso, el conjunto de imágenes final se dividirá en dos subconjuntos, uno de entrenamiento y otro de validación. Estos subconjuntos tendrán un 80 % y un 20 % de los datos respectivamente, por lo que dispondremos de 800 imágenes en el conjunto de entrenamiento y 200 en el conjunto de validación.

3.4.1. Primeros modelos

Inicialmente, los primeros modelos que construimos fue utilizando la arquitectura de la red Deconvnet introducida en la sección 2.1.4. Sin embargo, tras algunas pruebas, decidimos cambiar la arquitectura por una variante personalizada de la red U-Net también introducida en 2.1.4.

La principal razón de esta elección fue que U-Net es bastante sencilla de implementar y elimina las componentes de unpooling que aparecían en Deconvnet. Por tanto, solo hay que seguir un camino de reducción de dimensión usando bloques de convolución seguidos de bloques de pooling y, a continuación en el camino de recuperación de la dimensión se usan bloques de convolución transpuesta seguidos de capas de convolución. Además, en este proceso se usará la técnica de skip connections. El esquema de esta red se puede consultar en la figura 2.9.

Dado que finalmente implementaremos el modelo en una aplicación, necesitamos, aparte de una buena precisión, que el modelo ejecute rápidamente por lo que no nos va a interesar construir un modelo demasiado grande. Por ello, vamos a realizar ligeras modificaciones sobre la arquitectura anterior. La primera es que en todos nuestros modelos el tamaño de la imagen que entra a la red va a ser 128×128.

Primer modelo

Para nuestro primer modelo, además, redujimos el números de bloques de convolución en uno y el número de filtros que se aplican en cada capa. En particular, tras cada bloque formado por dos capas de

convolución en la fase de codificación, el número de filtros usado en el siguiente bloque se multiplicará por dos hasta acabar esta fase. Posteriormente, en la fase de decodificación se realiza el proceso inverso cambiando el número de filtros usados tras cada capa de convolución transpuesta a la mitad. En nuestro caso, la primera capa tiene 16 filtros. El esquema de la arquitectura de la red se puede ver en la siguiente figura.

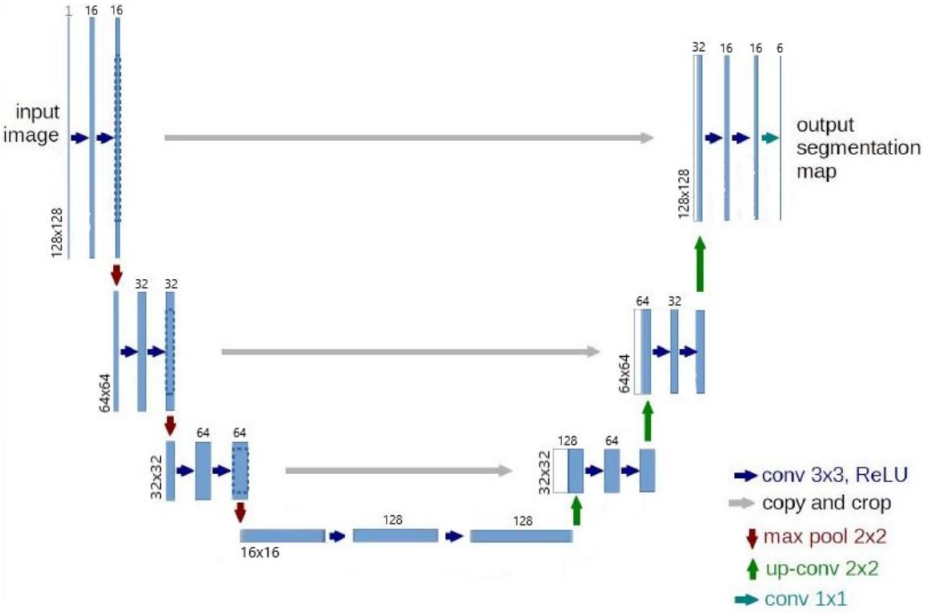


Figura 3.15: Arquitectura del primer modelo.

Como podemos ver, cada bloque en la fase de codificación está formado por dos bloques de convolución con filtros de tamaño 3×3, stride $= 1$ y función de activación ReLU. Tras cada bloque se aplica un max pooling de tamaño 2×2 que disminuirá el ancho y el alto de la imagen a la mitad. Además, en cada capa de convolución usaremos un padding de dimensión 1. La información de la red se va propagando hasta llegar al cuarto bloque de convoluciones que será cuando empiece el camino inverso para recuperar la dimensión original de la imagen de entrada.

Para esta tarea, tenemos tres bloques formados por una capa donde se aplica la convolución transpuesta con filtros de tamaño 2×2 y un stride $= 2$. Además, en este modelo implementamos la técnica de skip connections por lo que en cada bloque tras esta operación se concatenan los mapas de características. En cada bloque, a esta capa le sigue una capa de convolución con filtros de tamaño 3×3 y función de activación ReLU. Por último, en el bloque final tras la última capa de convolución se aplica una última capa de convolución con un número de filtros igual al número de clases y de tamaño 1×1. En esta última capa se usa la función de activación softmax para obtener las probabilidades de pertenencia del pixel a una de las clases. El número total de parámetros del modelo es 573.622.

Al compilar el modelo, la función de pérdida utilizada será la función de entropía cruzada en su versión categórica y el optimizador utilizado será el introducido en este trabajo (Adam).

Inicialmente el modelo se entrenó con 40 épocas, aunque mirando las gráficas de la figura 3.16 podemos ver que a partir de la época 30, la función de pérdida en el conjunto de validación no mejora el valor mínimo obtenido en dicha época. Por otro lado, en la gráfica de precisión del conjunto de validación, uno de los valores más altos se obtiene también en la época 30. A partir de ahí, en la gráfica aparecen picos lo que indica grandes saltos en la precisión del modelo, por lo que, para evitar que esta

varianza afecte a la precisión final de nuestro modelo convendrá entrenar el modelo con un número de épocas entre 26 y 30. Otra razón para elegir este número de épocas es el hecho de que a partir de la época 30, las gráficas de precisión en ambos grupos se empiezan a separar lo que puede indicar un posible sobreajuste del modelo a los datos de entrenamiento. Por tanto, para evitar este posible sobreajuste del modelo y dado que en la época 30 se consigue una de las precisiones más altas en el conjunto de validación, volvemos a entrenar el modelo con 30 épocas.

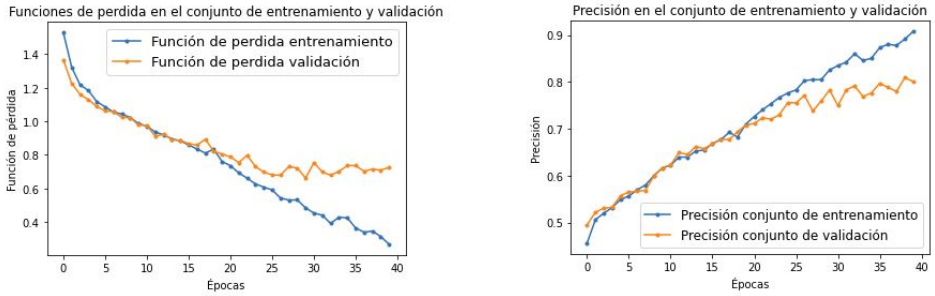


Figura 3.16: Gráficas de la evolución del error y la precisión en el conjunto de entrenamiento y validación para el modelo de 40 épocas.

El tiempo de entrenamiento del modelo con 30 épocas ha sido de 33 minutos y 53 segundos obteniendo una precisión de 82,36 % en el conjunto de entrenamiento y de 78,28 % en el conjunto de validación. Estas dos métricas miden la precisión por pixel por lo que, aunque sean buenas medidas, conviene tener otra herramienta que permita evaluar de forma más robusta el modelo. La herramienta que vamos a utilizar va a consistir en representar gráficamente la segmentación de las predicciones que obtenemos al probar el modelo con las imágenes del conjunto de validación. Con esto podemos observar visualmente la calidad de la segmentación que produce nuestro modelo.

Al igual que hicimos en la figura 3.14 coloreamos cada pixel de la imagen predicha en función de la clase a la que haya sido asignado. El código de colores para estas imágenes y el que usaremos de ahora en adelante va a ser el siguiente:

- Rojo: Pixeles que hayan sido clasificados como fondo.
- Amarillo: Pixeles que hayan sido clasificados como camiseta.
- Verde: Pixeles que hayan sido clasificados como pantalón.
- Azul claro: Pixeles que hayan sido clasificados como vestido.
- Morado: Pixeles que hayan sido clasificados como shorts.
- Rosa: Pixeles que hayan sido clasificados como zapatos.

Como podemos observar en la figura 3.17, el modelo consigue reconocer a la perfección las formas que tienen las prendas de ropa, sin embargo a la hora de clasificar los pixeles que corresponden a la prenda hay una mayor confusión. Por ejemplo, en la primera imagen vemos que la mayoría de pixeles del short se clasifican como pantalón lo cual puede tener cierto sentido puesto que ambas prendas son similares. Por otro lado, en la imagen de la camiseta vemos algunos pixeles que se han clasificado también incorrectamente como vestido. Por último, la segmentación de los zapatos es casi perfecta teniendo muy pocos pixeles mal clasificados.

Para hacernos una idea general de la confusión que tiene el modelo al diferenciar entre ciertas clases podemos examinar la matriz de confusión. En este caso, vamos a presentar dos matrices de confusión.

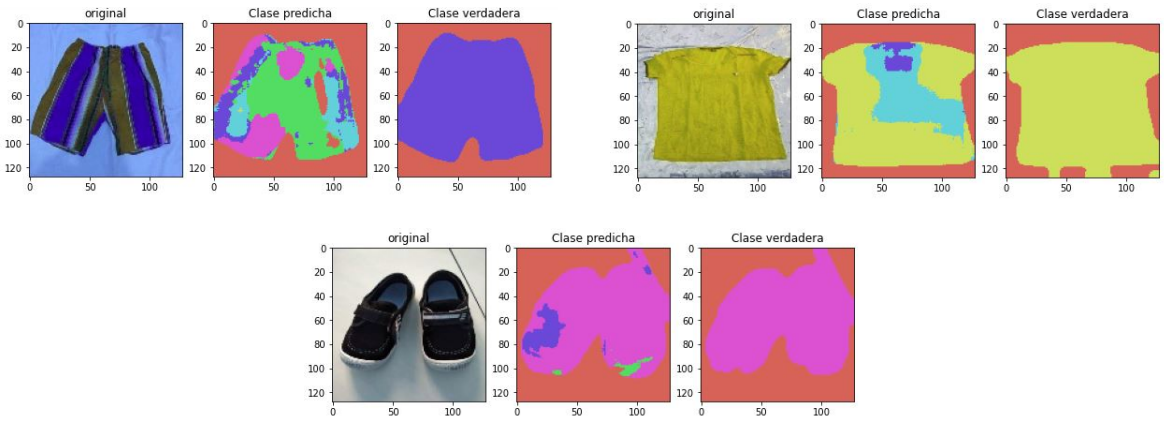
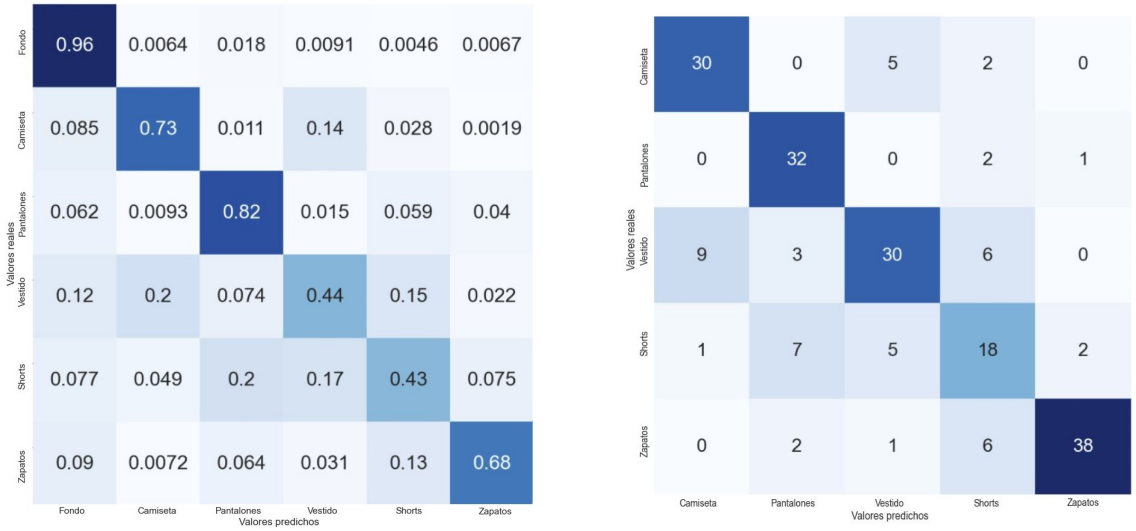


Figura 3.17: Imágenes originales, segmentación del modelo y etiqueta coloreada de tres imágenes del conjunto de validación.

La primera será una matriz normalizada de la clasificación a nivel de pixel y en la segunda veremos la matriz de confusión de la clasificación a nivel de prenda de ropa.



(a) Matriz de confusión a nivel de pixel.

(b) Matriz de confusión de la clasificación como prenda de ropa.

Figura 3.18: Matrices de confusión del primer modelo.

De la primera matriz de confusión podemos destacar la baja precisión en las clases short y vestido donde solo el 44 % de los pixeles se han clasificado correctamente. Por otro lado, también vemos cómo los pixeles mal clasificados de la clase vestido suelen haber sido clasificados en su mayoría como camisetas (20 %) mientras que en los shorts, la mayoría de pixeles mal clasificados han ido a parar a la clase de pantalones (20 %). Por tanto, podemos deducir que, debido a la posible similitud entre las prendas, los vestidos suelen ser confundidos como camisetas y viceversa y, los shorts como pantalones. Para confirmar esta hipótesis podemos mirar a la segunda matriz de confusión donde de los 33 shorts, 7 se han clasificado como pantalones y de los 48 vestidos, 9 se han clasificado como camisetas. Por último, en esta matriz también observamos cierta confusión entre shorts y vestidos y shorts y zapatos.

Para finalizar el análisis, podemos examinar la métrica que habíamos definido nosotros y que, en la sección 2.3.1, habíamos denominado índice de prendas bien clasificadas. En este modelo este índice es 0,74, es decir el 74 % de las prendas de ropa se han clasificado correctamente.

Segundo modelo

Con el análisis realizado anteriormente vimos que el modelo era claramente mejorable. Sobre todo había que centrarse en intentar mejorar la segmentación de las clases conflictivas del modelo que eran los vestidos y los shorts. Por tanto, la idea que decidimos llevar a cabo fue incluir un bloque de capas de convolución más en la fase de codificación y decodificación con el objetivo de que la red fuese capaz de extraer más características reduciendo aún más las dimensiones de la imagen y así ver si conseguía distinguir mejor entre esas clases conflictivas. Por tanto, en las capas más profundas, la arquitectura de la red queda de la siguiente forma:

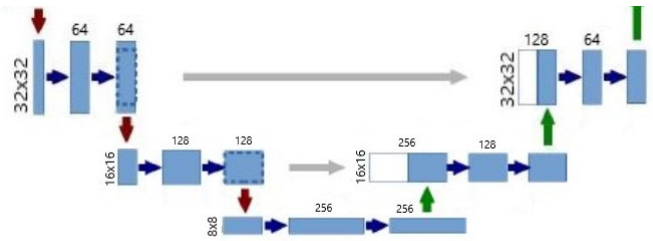


Figura 3.19: Arquitectura en las capas profundas del segundo modelo.

El resto del modelo es igual que el anterior y se van a usar los mismo valores de padding y del stride en las capas de convolución y de deconvolución. A la hora de compilar el modelo, también se ha usado la misma función de pérdida y el mismo optimizador. En total, este modelo tiene 1.721.720 parámetros.

Al igual que con el modelo anterior, este se ha entrenado inicialmente con 35 épocas obteniendo las gráficas de evolución de la función de pérdida y de precisión en los conjuntos de entrenamiento y validación de la figura 3.20. Como podemos ver en dichas gráficas de las funciones de pérdida, en la época 25, la función de pérdida en el conjunto de validación alcanza su mínimo y, a partir de ese momento, el valor empieza a aumentar sin volver a descender en las siguientes épocas. Este hecho se puede observar también en la gráfica de precisión en el conjunto de validación donde, a partir de la época 25, la precisión tiende a mantenerse constante. Por otro lado, a partir de esta época la precisión en el conjunto de entrenamiento sigue aumentando por lo que parece que a partir de esa época el modelo comienza a sobreajustar. Por tanto, al igual que hicimos antes vamos a entrenar el modelo de nuevo con 25 épocas.

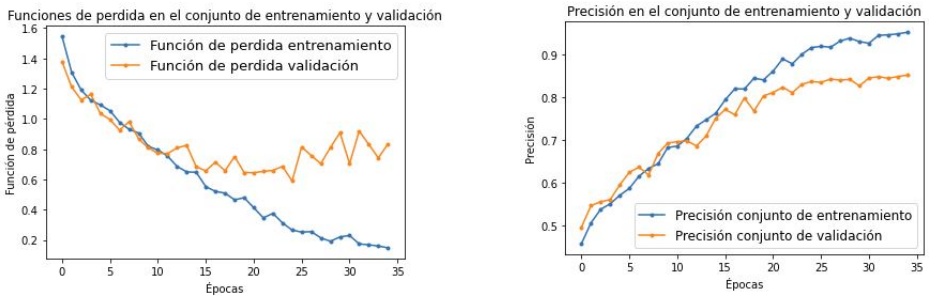


Figura 3.20: Gráficas de la evolución del error y la precisión en el conjunto de entrenamiento y validación del segundo modelo con 35 épocas.

Este nuevo modelo con 25 épocas ha tenido un tiempo de entrenamiento de 32 minutos y 45 segundos. La precisión obtenida en el conjunto de entrenamiento ha sido 91,02 % y en el conjunto de validación ha sido 83,63 %. Con solo estas dos medidas ya vemos que a nivel de precisión de pixel este modelo supera claramente al anterior.

A continuación, vamos a comparar la segmentación conseguida por el modelo en las imágenes del conjunto de validación. Con el objetivo de poder comparar entre ambos modelos, vamos a representar las mismas predicciones que en el modelo anterior.

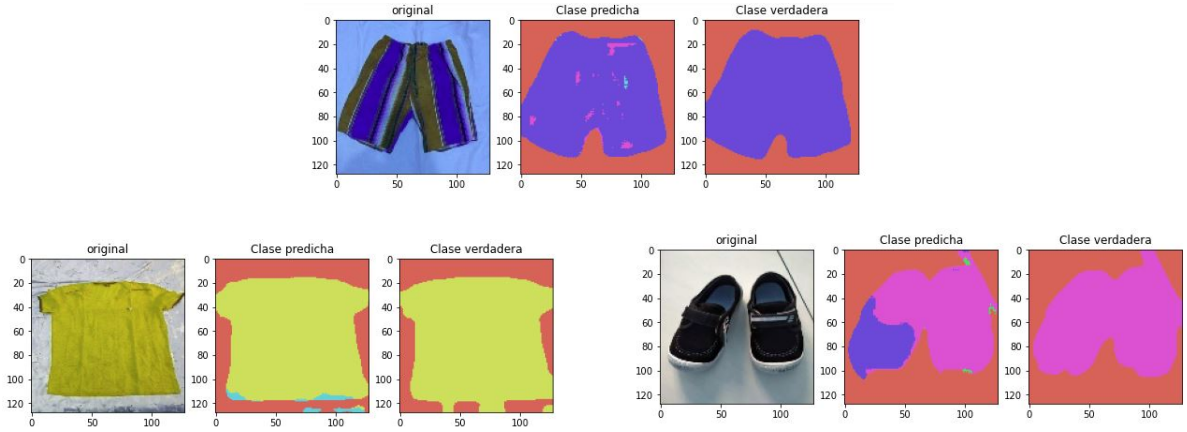


Figura 3.21: Imágenes originales, segmentación del modelo y etiqueta coloreada de tres imágenes del conjunto de validación del segundo modelo.

Como podemos observar, las predicciones del short y de la camiseta han mejorado sustancialmente. De hecho, muy pocos pixeles se han clasificado incorrectamente. Por otro lado, sí es cierto que la segmentación de los zapatos empeora ligeramente, aunque en conjunto sigue siendo buena. Una forma de validar la segmentación comparándola con la etiqueta original es hacer uso de la métrica IoU introducida en la sección 2.3.1. Como vimos, está métrica permitía hacer la ratio entre el área de superposición de las etiquetas y las segmentaciones y el área de la unión de ambas imágenes. Por tanto, vamos a analizar los resultados de esta métrica por clase y el valor de su media en la siguiente tabla:

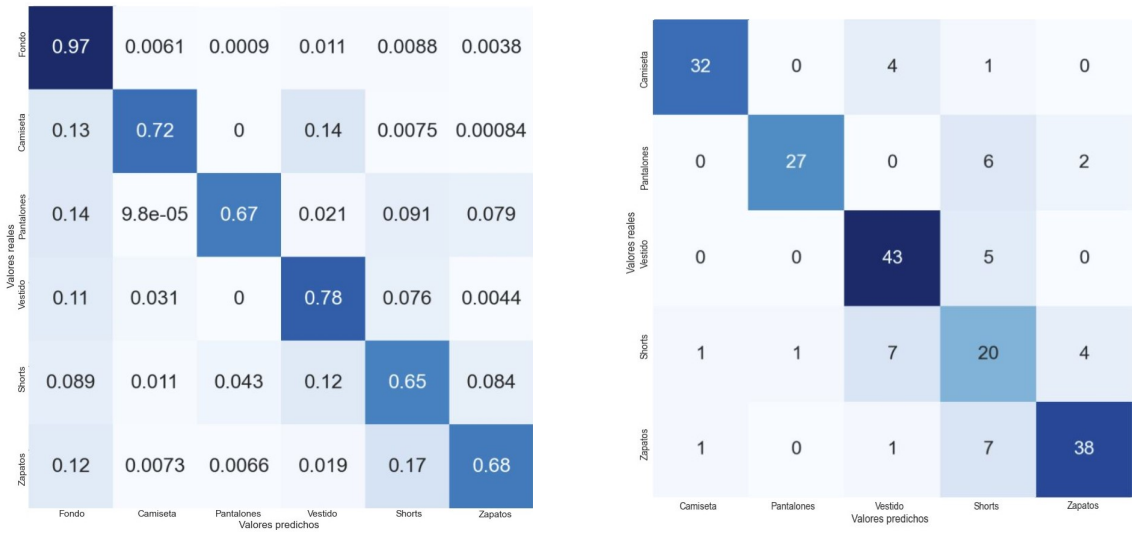
Clase	TP	FP	FN	IoU
Fondo	1578726	193953	49550	0.866
Camiseta	213665	28584	83924	0.655
Pantalón	213740	15123	103725	0.643
Vestido	326109	105883	93272	0.621
Shorts	170224	137426	91543	0.426
Zapatos	238001	55366	114321	0.584
MIoU = 0.633				

Cuadro 3.1: Tabla donde en cada clase se presentan los verdaderos positivos, los falsos positivos, los falsos negativos y la métrica IoU.

En general para las cuatro primeras clases tenemos un valor bueno de la métrica Iou, sobre todo para los pixeles de la clase fondo que, como se puede intuir, son los que mejor clasifica el modelo. Sin embargo, destaca el valor de IoU de la clase shorts que es menor que 0,5 debido a la cantidad de falsos positivos de esta clase. Esto probablemente indique que los pixeles mal clasificados de las otras prendas hayan ido a parar, la mayoría, a la clase shorts como hemos visto por ejemplo en la segmentación del par de zapatos. Para la clase zapatos el valor del Iou no es malo pero es un poco inferior a la media.

Por último, el valor de MIoU es aceptable, por encima de 0,6 que claramente se ve afectado por el bajo valor del índice IoU en la clase shorts.

Para finalizar el análisis de este modelo y ver si la hipótesis de que la mayoría de pixeles mal clasificados de las prendas de ropas que no son shorts, son clasificados como shorts vamos a examinar las matrices de confusión. Al igual que con el modelo anterior presentaremos la matriz de confusión a nivel de pixel y la matriz de confusión de la clasificación por prenda de ropa:



(a) Matriz de confusión a nivel de pixel.

(b) Matriz de confusión de la clasificación como prenda de ropa.

Figura 3.22: Matrices de confusión del segundo modelo.

En la primera matriz de confusión, al contrario de como pensábamos, podemos ver cómo la mayoría de errores de clasificación en las cinco prendas de ropa son por la asignación de los pixeles de dichas clases a la clase fondo. De hecho, podemos ver cómo el porcentaje de pixeles mal clasificados como fondo varía entre el 8 % y el 14 % según la clase. Por otro lado, el porcentaje de pixeles de camisetas clasificados como vestidos sigue siendo del 14 % por lo que en este aspecto el problema no ha mejorado. Sin embargo, sí que se reduce la cantidad de pixeles que son vestido y se clasifican como shorts pasando de un 15 % a un 7 %, al igual que el problema que teníamos entre pantalones y shorts. Por último, seguimos teniendo el problema de los pixeles de imágenes de zapatos que se clasifican como shorts.

Por tanto, es cierto que el número de verdaderos positivos en las clases vestidos y shorts ha subido, un 34 % y un 22 % respectivamente lo cual es una mejora respecto del modelo anterior pero, en este caso tenemos el problema de que el modelo clasifica muchos pixeles de prendas de ropa como fondo.

En cuanto a la segunda matriz de confusión, en este caso sí que se aprecia más la confusión entre las diferentes prendas de ropa y los shorts. Solamente hay que fijarse en la columna de shorts donde se encuentran los falsos positivos. Además, igual que en el anterior modelo, podemos examinar la métrica de la precisión de acierto por prenda de ropa que en este caso es del 80 %.

Para terminar el análisis de este segundo modelo, podemos decir que este modelo es superior al anterior, no sólo en cuanto precisión por pixel, sino también en precisión por prenda de ropa y porque resuelve los problemas que teníamos con los shorts y los vestidos. Sin embargo, surge el nuevo problema de los pixeles mal clasificado como fondo. Por ello, la siguiente prueba que se hizo y que comentaremos brevemente fue ampliar el número de filtros que se usaban en cada bloque del modelo.

Variante del segundo modelo

Esta variante se diferencia de la anterior únicamente en el número de filtros usados en cada capa. En los resultados vamos a observar que el modelo no mejora mucho por lo que simplemente los comentaremos brevemente para que sirva como ejemplo de que, a la hora de construir una red, tan malo es quedarse corto de parámetros, dado que la red no generalizará, como sobreparametrizar en exceso puesto que, como veremos, los resultados que se obtienen son ligeramente peores.

Por tanto, esta red será idéntica que la anterior salvo que en el primer bloque el número de filtros de cada capa de convolución será 32 y no 16. De esta forma, en el segundo bloque se usarán 64 filtros y así se continuará hasta que en el quinto bloque se usen 512 filtros. Este cambio incrementa enormemente el número de parámetros de la red, pasando a ser 6.880.614.

Tras un estudio de la evolución de las gráficas de las funciones de pérdida y de las precisiones análogo al realizado para los anteriores modelos, este modelo se ha entrenado con 29 épocas, siendo el tiempo total de entrenamiento 1 hora 46 minutos y 53 segundos. Este tiempo es sustancialmente más grande que en los anteriores modelos, sin embargo, vamos a ver que este modelo no mejora los resultados obtenidos con el anterior.

Para comenzar, la precisión del modelo obtenido es 88 % en el conjunto de entrenamiento y 82 % en el conjunto de validación que son ligeramente inferiores a las del modelo anterior. Por otro lado, el valor del MIoU es 0,603 que también es inferior al del segundo modelo. En cuanto a la segmentación, si nos fijamos en los ejemplos de la figura 3.23 podemos ver cómo en los shorts la segmentación empeora, en la camiseta se mantiene parecida y en los zapatos mejora.

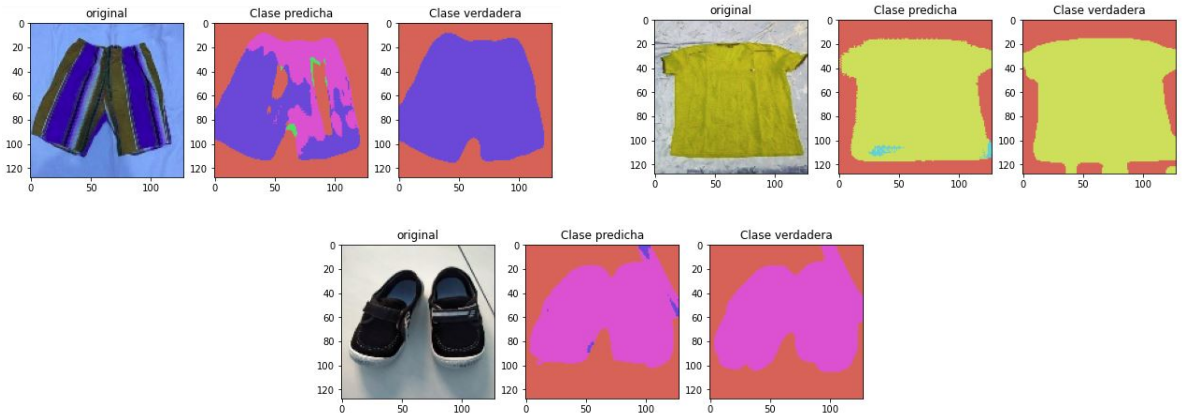
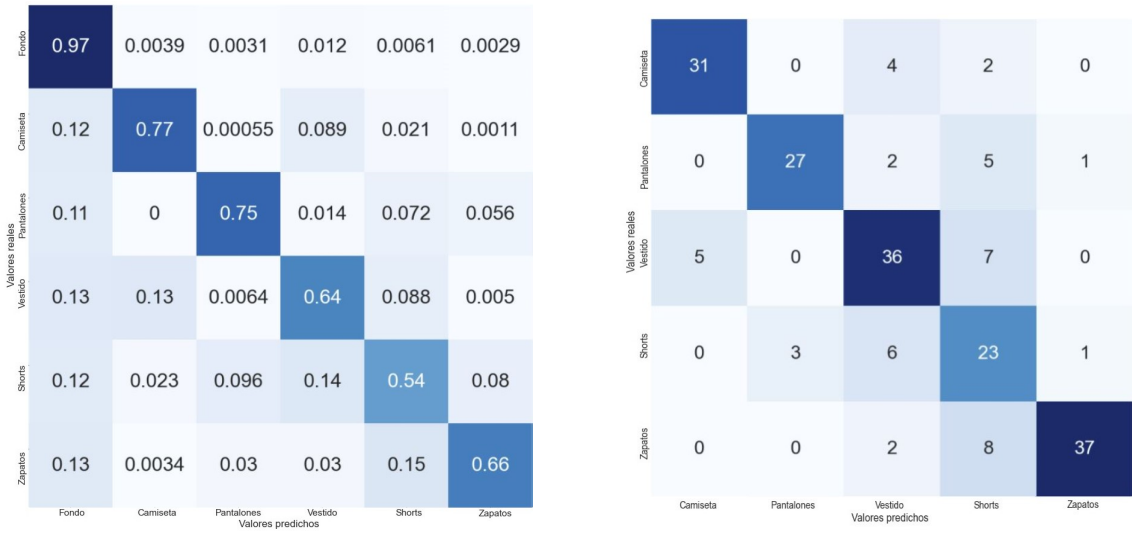


Figura 3.23: Imágenes originales, segmentación del modelo y etiqueta coloreada de tres imágenes del conjunto de validación de la variante del segundo modelo.

Para mirar el rendimiento de este modelo en general, podemos examinar las matrices de confusión del modelo en la figura 3.24. En la primera matriz de confusión destaca que solo el 54 % de los pixeles de los shorts se han clasificado correctamente, porcentaje que disminuye en un 14 % con respecto al modelo anterior. También, en las clases vestido y zapatos este porcentaje disminuye. Por otro lado, también podemos observar que se sigue manteniendo el problema de la asignación de pixeles que pertenecen a prendas de ropa a la clase fondo.

En cuanto a la segunda matriz, los valores son similares por lo que se observa poco cambio salvo en la clase vestido donde pasamos de 43 vestidos acertados a 36. Este cambio viene producido por los 5 vestidos que han sido clasificados erróneamente como camisetas. Esto hace que si calculamos la métrica de precisión de prenda de ropa en este modelo obtengamos que es del 77 %, 3 puntos por debajo del

modelo anterior.



(a) Matriz de confusión a nivel de pixel.

(b) Matriz de confusión de la clasificación como prenda de ropa.

Figura 3.24: Matrices de confusión de la variante del segundo modelo.

Para finalizar este análisis, hemos visto que el rendimiento del modelo es inferior que el anterior en casi todos los aspectos por lo que queda patente que el incrementar los parámetros de la red no siempre conlleva una mejora del modelo.

3.4.2. Modelo final

Hemos visto tres modelos con los que hemos conseguido resultados aceptables aunque todos tienen algunas limitaciones. Entre ellas, hemos visto la confusión en la clasificación entre algunas clases del modelo, sobre todo en los shorts, y la confusión entre los pixeles de las prendas de ropa y del fondo. Además, el valor de las métricas como la de MIoU es mejorable. Por tanto, el siguiente paso fue refinar el segundo modelo, ya que es el que mejores resultados obtiene, con el fin de mejorar las segmentaciones obtenidas. Para ello, se hicieron algunos pequeños cambios en la arquitectura, se cambiaron las funciones de activación y se probó con diferentes optimizadores a la hora de compilar del modelo. Estas pruebas no las incluimos en la memoria para no extender demasiado el trabajo. A continuación, vamos a mostrar el modelo que mejores resultados obtuvo tras todas estas pruebas.

Arquitectura y entrenamiento

La arquitectura de este último modelo que podemos observar en la figura 3.25 es muy parecida a la presentada en el segundo modelo. Seguimos teniendo en la fase de codificación 5 bloques de capas de convolución con 16, 32, 64, 128 y 256 filtros respectivamente. En las operaciones de convolución de estas capas se usa un stride $= 1$ y un padding de dimensión 1. Lo que varía en esta arquitectura es que la función de activación utilizada tras las convoluciones es la función ELU. Al igual que antes, tras cada bloque se aplica un max pooling de tamaño 2×2.

La fase de decodificación es análoga a la del segundo modelo, cada bloque incluye una capa de convolución transpuesta seguida de dos capas de convolución. Además, se utiliza la técnica skip connections donde se concatenan los mapas de características tras la capa de convolución transpuesta en cada bloque. Por último, la última capa es una capa de convolución de tamaño 1×1 y tiene un número de filtros igual al número de clases del modelo. Tras esta capa se aplica la función de activación softmax

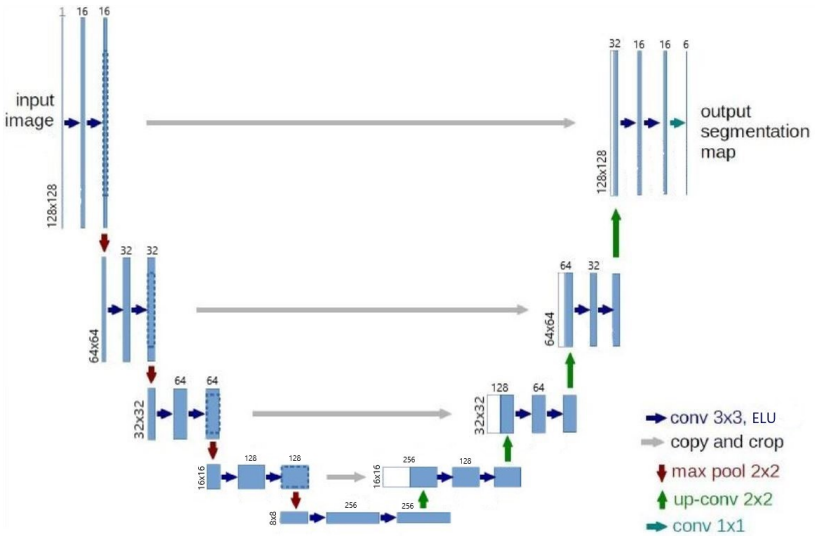


Figura 3.25: Arquitectura del modelo final.

para obtener las probabilidades de pertenencia de cada pixel a cada una de las clases del modelo. Este modelo tiene un total de 1.721.270 parámetros.

Para poder evaluar la evolución de las precisiones y de las funciones de pérdida del modelo en los conjuntos de entrenamiento y validación decidimos entrenar inicialmente el modelo con 35 épocas y a partir de ahí ajustar este número en función de los resultados.

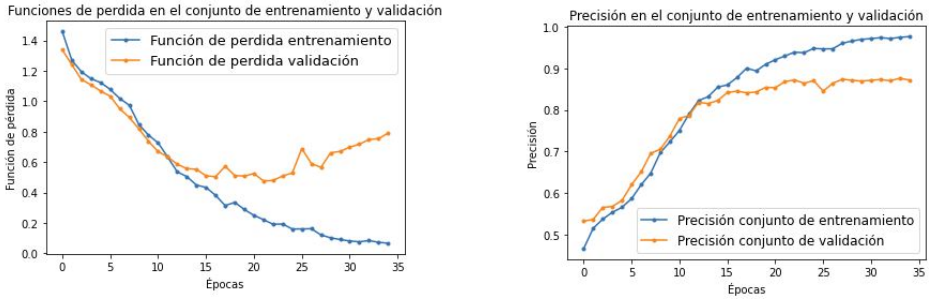


Figura 3.26: Gráficas de la evolución del error y la precisión en el conjunto de entrenamiento y validación del modelo final con 35 épocas.

En la primera gráfica vemos cómo a partir de la época 17-18 las gráficas de las funciones de pérdida de ambos conjuntos se empiezan a distanciar y ya no decrecen uniformemente. De hecho, la función de pérdida del conjunto de validación alcanza su mínimo en la época 22 y a partir de ahí la función vuelve a crecer. Por otro lado, en las gráficas de precisión ambas gráficas se empiezan a separar entorno a la época 18 y, en este caso, es en la época 25 donde se alcanza la mayor precisión en el conjunto de validación. A partir de ahí, la precisión tiende a mantenerse constante mientras que la precisión en el conjunto de entrenamiento sigue aumentando.

Por tanto, en este caso tenemos dos criterios para elegir en que época detener el entrenamiento. El primero es atendiendo al valor de la función de pérdida y el segundo al valor de la precisión, ambos en el conjunto de validación. En nuestro caso se ha elegido que prime el valor de la precisión ya que el valor de la función de pérdida no crece demasiado en 2 épocas. Por tanto, volvemos a entrenar el modelo para 25 épocas.

Para 25 épocas obtenemos las gráficas de evolución de la figura 3.27 y, el modelo que obtenemos tiene un 94,37 % de precisión en el conjunto de entrenamiento y un 87,04 % en el conjunto de validación. Ambas medidas superan a las obtenidas con los modelos anteriores. El tiempo de entrenamiento de este último modelo ha sido 39 minutos y 10 segundos.

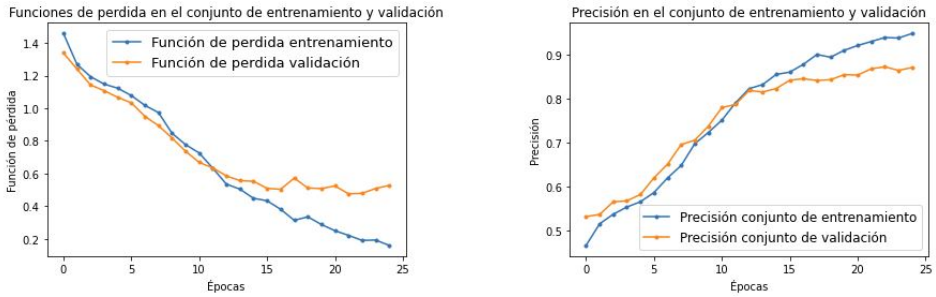


Figura 3.27: Gráficas de la evolución del error y la precisión en el conjunto de entrenamiento y validación del modelo final con 25 épocas.

Validación del modelo

Para validar el modelo vamos a proceder como con los modelos anteriores. Haremos un análisis visual de las predicciones del modelo para algunas imágenes del conjunto de validación, examinaremos alguna de las métricas más importantes y las matrices de confusión del modelo y finalizaremos con el análisis de la curva ROC. Comencemos entonces mirando, en la siguiente figura, alguna de las predicciones obtenidas por el modelo.

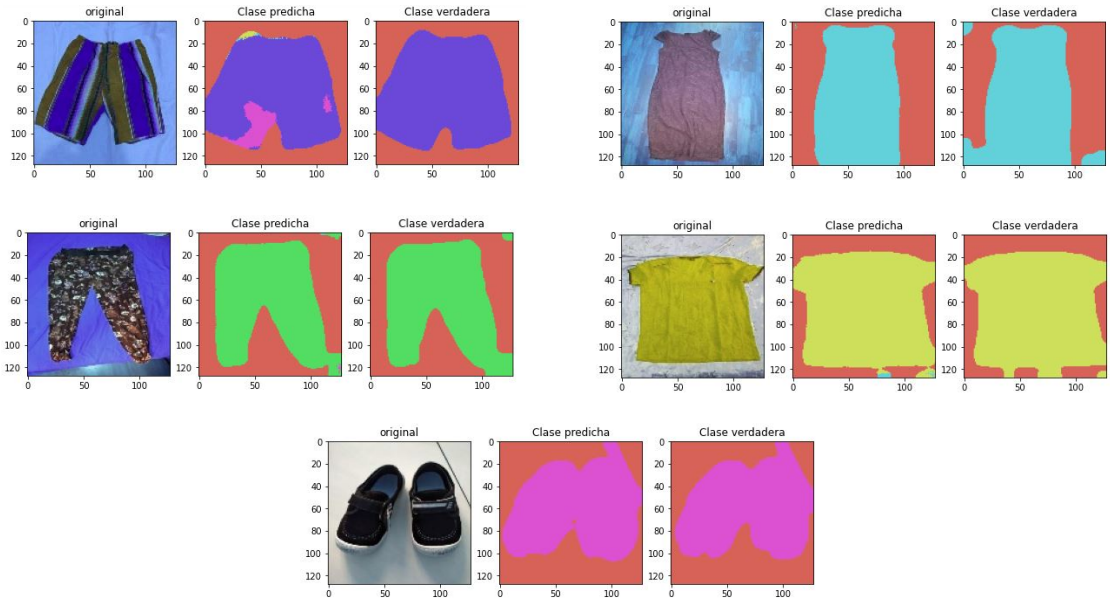


Figura 3.28: Imágenes originales, segmentación del modelo final y etiqueta coloreada de cinco imágenes representativas de cada clase del conjunto de validación.

Como podemos ver, la segmentación que realiza el modelo de las 5 prendas de ropa es bastante buena salvo en la del short que tiene algún error en la parte inferior. Además, si miramos por ejemplo la segmentación del vestido vemos que mejora la segmentación realizada en la etiqueta ya que reconoce únicamente la forma de la prenda de ropa. Al igual que hicimos antes, vamos a examinar los valores de la métrica IoU por clase y la media de esos valores en la siguiente tabla:

Clase	TP	FP	FN	IoU
Fondo	1537481	106097	90795	0.886
Camiseta	248308	66040	49281	0.683
Pantalón	253749	40604	63716	0.709
Vestido	325735	71042	93646	0.664
Shorts	201006	106667	60761	0.546
Zapatos	285861	34210	66461	0.740
MIoU = 0.705				

Cuadro 3.2: Tabla donde en cada clase se presentan los verdaderos positivos, los falsos positivos, los falsos negativos y la métrica IoU.

Si comparamos los valores con los obtenidos en el segundo modelo, destaca el índice obtenido en la clase zapatos que se ha incrementado en casi 0,2. En el resto de clases el valor del índice también es bueno aunque algo peor en la clase shorts. En cuanto a la media general, un valor por encima de 0,7 suele ser considerado como indicador de que la segmentación realizada por el modelo ha sido buena. Con esta información podemos prever que las matrices de confusión del modelo habrán mejorado.

(a) Matriz de confusión a nivel de pixel.

(b) Matriz de confusión de la clasificación como prenda de ropa.

Figura 3.29: Matrices de confusión del modelo final.

Como podemos ver en la primera matriz, en todas la clases los verdaderos positivos superan el 75 % de los pixeles lo cual mejora en gran cantidad los resultados obtenidos por los modelos anteriores. En cuanto a los errores de clasificación vemos que depende de las clases que observemos. Por ejemplo, el modelo suele confundir las camisetas con vestidos y viceversa como podemos ver por ese 8 % de falsos positivos entre esas clases. Lo mismo podríamos decir de pantalones y shorts donde, en este caso, hay valores de falsos positivos cercanos al 9 %. Por otro lado, también hay cierta confusión entre la clase shorts y la clase zapatos. Por último, destacar que este modelo mejora también el porcentaje de pixeles de prendas de ropa que se han clasificado como fondo, no superando en ningún caso el 8 %.

Estos errores que comete el modelo son entendibles si analizamos las diferentes clases del modelo. Por ejemplo un vestido que tenga en la parte superior mangas más largas que en la figura 3.28 puede

confundirse con una camiseta. De la misma manera, un pantalón y un short son similares en la forma de la prenda. Lo que puede generar más dudas es la clasificación de algunos zapatos como shorts. Sin embargo, si miramos la etiqueta del zapato de la figura 3.28 y no nos fijamos en la imagen original, vemos que la forma de la etiqueta es parecida a un short ya que los zapatos no aparecen separados. Por tanto, esta es la principal razón por la cual algunos zapatos los clasifica como shorts.

En cuanto a la segunda matriz, vemos que en camisetas, vestidos y zapatos hay muy pocos errores 2, 4 y 3 respectivamente. De hecho, los errores en vestidos y zapatos vienen de la clasificación de algunas prendas como shorts. En cuanto a pantalones y shorts sí que observamos más errores aunque la mayoría suelen ser por la confusión entre esas dos clases. En particular, 5 pantalones que han sido clasificados como shorts y 3 shorts que han sido clasificados como pantalones. Por tanto, en general vemos que la clasificación por prenda de ropa de este modelo también supera con creces a la de modelos anteriores. De hecho, en este modelo la métrica de precisión por prenda de ropa es 90 %.

Para finalizar el análisis vamos a analizar la gráfica de la curva ROC del modelo. En este proyecto hemos programado dos tipos de gráfica para calcular la curva ROC con el objetivo de realizar un análisis más completo. La primera será la curva de la clasificación a nivel de pixel realizada por el modelo y la segunda nos permitirá analizar la curva de la clasificación a nivel de prenda de ropa del modelo. Además, en ambas gráficas se han enfrentado la clase positiva correspondiente vs. el resto en vez de realizar un análisis por pares.

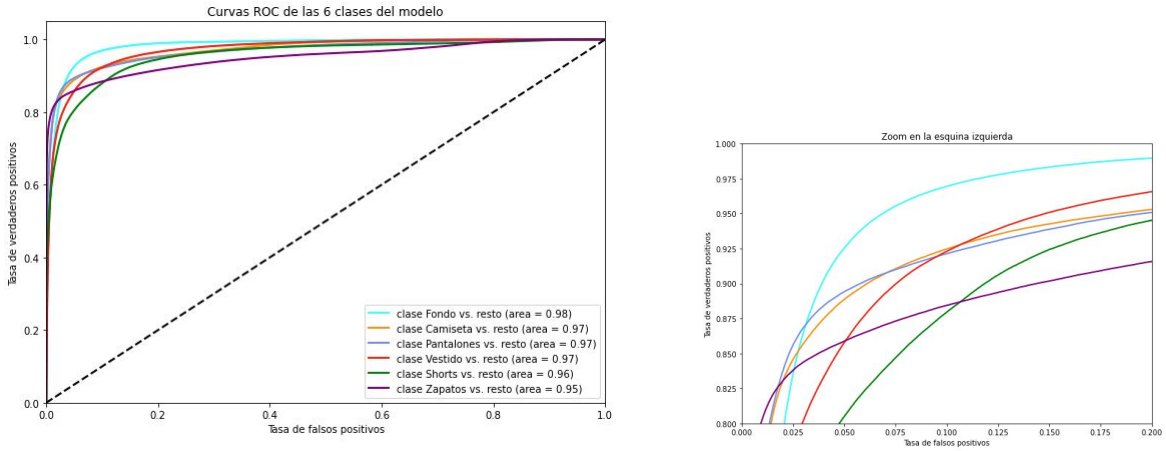


Figura 3.30: Curvas ROC a nivel de pixel y zoom en la esquina izquierda del gráfico.

En este primer gráfico vemos cómo las curvas de las 6 clases del modelo crecen rápidamente hasta llegar a 0,8. A partir de ese punto, las curvas comienzan a aplanarse. En el gráfico de la derecha podemos ver la evolución de las curvas con más claridad. Queda claro que la curva ROC de la clase fondo supera al resto y así nos lo indica el valor del AUC, 0,98. Por otro lado, las curvas de las clases camiseta, pantalón y vestido tienen una trayectoria parecida y el AUC es el mismo para las tres. En cuanto a la clase shorts, la curva no crece rápidamente al inicio, sin embargo según aumenta la tasa de falsos positivos esta llega a igualar a las de las tres clases anteriores. Todo lo contrario que la curva de la clase zapatos, la cual crece rápidamente al inicio para después aplanarse, lo que conlleva que esta curva tenga el menor AUC.

En vista de los valores obtenidos de AUC se podría decir que el modelo tiene un rendimiento excelente pues todos están por encima de 0,95. Sin embargo, hay que recordar que este es el gráfico de las curvas ROC a nivel de pixel. De hecho, si nos fijamos en los valores de verdaderos positivos y falsos positivos del cuadro 3.2 veremos que los primeros son mucho más grandes que los segundos lo que hace que al representar gráficamente las tasas de verdaderos positivos y de falsos positivos obtengamos curvas que crecen tan rápidamente. Por tanto, una forma de asegurarnos también de que el rendimiento

del modelo es bueno, es analizar las curvas ROC del modelo según la clasificación como prenda de ropa.

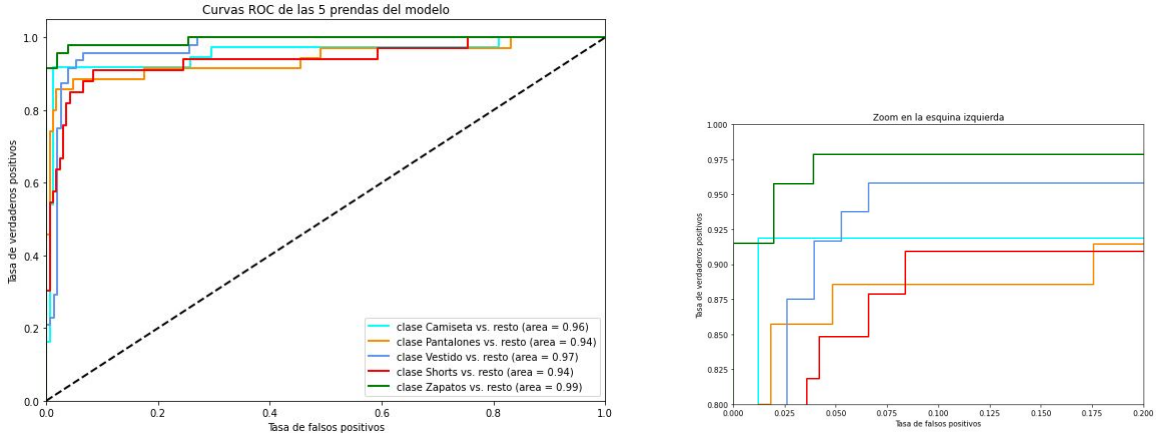


Figura 3.31: Curvas ROC según la clasificación de la prenda y zoom en la esquina izquierda del gráfico.

Con esta gráfica podemos concluir que el modelo tiene un rendimiento excelente en la clasificación de la clase zapatos, hecho que ya habíamos comentado en la matriz de confusión de la clasificación por prenda de ropa. Por otro lado, también vemos que el AUC de las curvas de las clases pantalones y shorts es menor que el del resto de clases por lo que la clasificación en estas clases es un poco peor. Sin embargo, en general todas las curvas tienen un AUC mayor que 0,9 por lo que podemos concluir que el rendimiento de nuestro modelo es bastante bueno.

Por último, comentar que aparte de validar el modelo usando las diferentes métricas comentadas anteriormente, también se utilizó un conjunto test formado por 25 imágenes sacadas de la web de venta de segunda mano online wallapop, [32]. Esta web permite subir fotos de los productos que se quieren vender sacadas por los usuarios, por lo que las imágenes de prendas de ropa son similares a las usadas en nuestro conjunto de datos. Algunos de los resultados del modelo en estas imágenes son los siguientes:

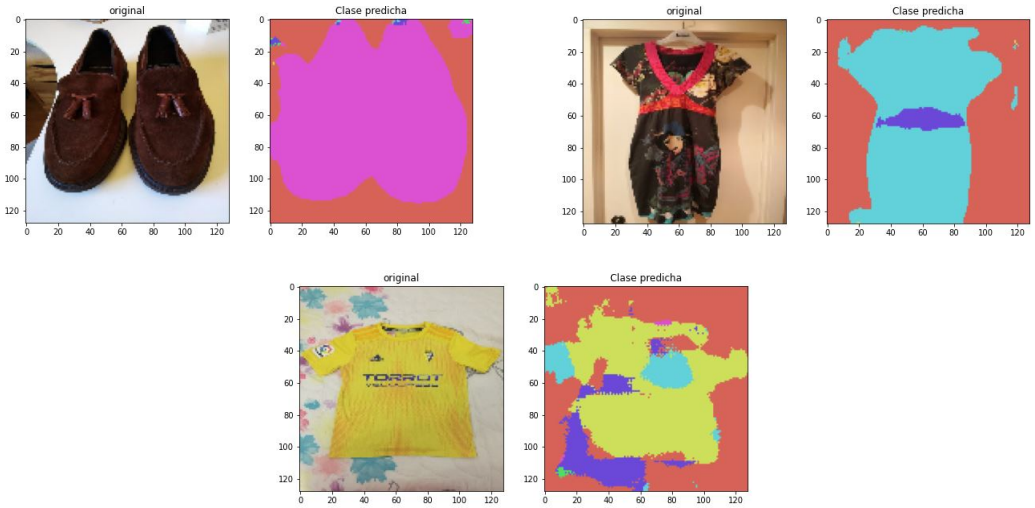


Figura 3.32: Tres ejemplos de la segmentación que realiza el modelo con imágenes de wallapop.

Podemos ver cómo en el conjunto test, las segmentaciones obtenidas por el modelo siguen siendo buenas aunque en la camiseta hay varios conjuntos de pixeles mal clasificados debido probablemente al estampado que aparece en el fondo de la imagen.

Limitaciones del modelo

Hasta ahora hemos visto que el modelo actúa bastante bien con imágenes donde solo hay una prenda de ropa. Sin embargo, dado que vamos a construir un sistema de recomendación y una aplicación que implementen este modelo, debemos ser conscientes de sus limitaciones. Para poner a prueba nuestro modelo hemos utilizado imágenes donde aparecen varias prendas de ropas juntas, sacadas de [32], e imágenes donde aparecen personas con varias prendas de ropa.

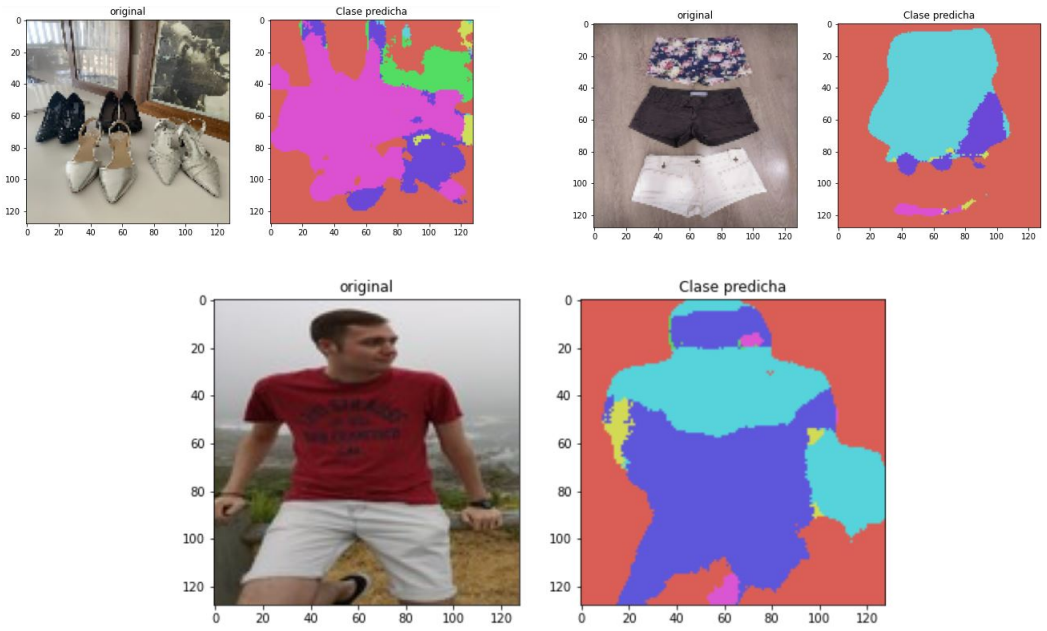


Figura 3.33: Tres ejemplos de las limitaciones del modelo.

En la primera imagen de la figura 3.33 vemos cómo el modelo reconoce que los objetos de la imagen son zapatos pero el modelo no es capaz de definir sus formas para separarlos. En la segunda imagen pasa algo parecido, como tenemos varios shorts juntos, el modelo interpreta que son una única prenda y los clasifica como si fuesen un vestido. Por último, en imágenes con personas vemos como el modelo es capaz de identificar la forma correctamente la zapatilla y los shorts. Sin embargo, dado que en la imagen no hay una distinción clara de dónde termina el short y empieza la camiseta, el modelo predice que el short es más largo de lo que realmente es. Además, hace una clasificación errónea de la camiseta, clasificándola como vestido. Por último, notar también que la cara y los brazos no se clasifican en la clase fondo.

Estos errores son entendibles, pues hay que recordar que nuestra red se ha entrenado con imágenes de prendas de ropa sueltas y no todas tenían una calidad buena. Por ello, es normal que cuando aparezcan elementos que la red no ha visto nunca, como una persona, cometa errores de clasificación. Por ello también, el sistema de recomendación construido irá enfocado sobre todo a imágenes parecidas a las que hemos usado en el conjunto de entrenamiento o en el conjunto test.

3.5. Sistema de recomendación

Una vez que tenemos un modelo lo suficientemente bueno, ya podemos implementarlo para construir el sistema de recomendación y la aplicación final. Como ya comentamos en la introducción del trabajo, nuestro sistema de recomendación va a tener dos opciones que podrá elegir el usuario. En la primera se recomendarán prendas del mismo tipo y de los mismos colores que la de la imagen subida y en la segunda simplemente prendas del mismo tipo. En cada una de las recomendaciones se incluirá un

total de seis prendas recomendadas.

Para poder tener un sistema de recomendación eficiente se necesita también disponer de una base de datos lo suficientemente buena de la cual obtener las imágenes que se van a recomendar. En nuestro caso, para crear la base de datos vamos a usar el archivo excel que venía con el conjunto de datos de [13]. En este archivo disponemos del nombre de la imagen y del tipo de prenda que es, por lo que sólo nos faltaría ampliarla para tener el color de cada prenda de ropa. Este proceso lo explicamos en la siguiente sección.

3.5.1. Creación de la base de datos.

Dado que ir examinando cada prenda de ropa que hay en la hoja excel para obtener el color sería un proceso que llevaría mucho tiempo, en esta sección proponemos un proceso automatizado para obtener el color predominante de la prenda. En este algoritmo se va a hacer uso del algoritmo k-medias introducido en la sección 3.3.1 que, como vimos, en el caso de una imagen, agrupaba los pixeles según su color e intensidad.

Por tanto, cada cluster creado por el algoritmo tendrá en su grupo pixeles con colores similares al pixel que sea el centroide de dicho cluster. De esta forma, los diferentes centroides de cada cluster serán, a priori, los colores más predominantes de la imagen. Para obtener el color que más aparece en la imagen, simplemente sumamos la cantidad de pixeles en cada cluster y nos quedamos con el centroide del cluster que más pixeles tenga.

Como estamos trabajando con una imagen a color, este centroide será un vector de longitud 3 donde cada componente representa cada uno de los colores de la codificación rgb. Por tanto, este centroide representa un color. Luego, para escribir el color en la base de datos, necesitamos conocer el color que representa dicha tupla de tres valores.

Para esta tarea, hemos creado una función la cual, dada una tupla en formato rgb, busca el color que corresponde a dicha tupla en un diccionario que incluye 140 colores, [14]. Esta búsqueda se realiza haciendo uso de una función llamada KDTree del paquete scipy.spatial de Python la cual devuelve el vecino del diccionario de colores más cercano a la tupla que se le pasa como parámetro. Para más información sobre el algoritmo usado por esta función se puede consultar [22].

Por otro lado, también vamos a aprovechar el hecho de que las prendas de ropa se encuentren en el centro de las imágenes. Antes de aplicar el algoritmo k-medias, haremos un zoom a la imagen de forma que podamos asegurarnos de que el color predominante que obtengamos corresponda a la prenda de ropa y no al fondo. Vamos a ilustrar el proceso descrito aplicándolo a la imagen de la figura 3.12.

Primero, redimensionamos la imagen a tamaño 128×128 y hacemos zoom de forma que recortamos 28 pixeles de cada lado de la imagen. A continuación, aplicamos el algoritmo de k-medias con 4 clusters para asegurarnos formar un número de grupos suficientes para recoger los colores que aparecen en la imagen. Con la información devuelta por el algoritmo calculamos el porcentaje de pixeles que pertenecen a cada cluster sobre el total.

El gráfico de sectores con los porcentajes de la figura 3.34 indican que el cluster 0 tiene el 46,7 % de los pixeles de la imagen por lo que el centroide de dicho cluster es el color predominante de la imagen y, por tanto, el color de la prenda de ropa. Los valores en la escala rgb del centroide de dicho cluster son [86,78250103, 42,83408997, 160,13867107]. Si pasamos esta tupla a la función que hemos descrito anteriormente, obtenemos que el color de la prenda de ropa es *darkslateblue*.

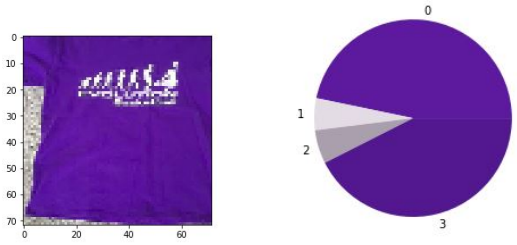


Figura 3.34: Zoom de la imagen y gráfico de sectores donde se puede apreciar el porcentaje de aparición de cada color.

Este proceso que hemos descrito con detalle se aplica a todas las imágenes que tenemos en el conjunto de datos del problema y en nuestra nueva base de datos se apuntan el nombre de la imagen, el tipo de prenda y el color obtenido mediante este procedimiento.

3.5.2. Recomendación en función de la prenda de ropa

Una vez definida la base de datos a través de la cual vamos a coger las prendas a recomendar, vamos a describir primero cómo se realiza el proceso de la opción de recomendación en función de la prenda de ropa que se suba al sistema.

Para que sea más sencillo de entender, vamos a explicarlo sobre un ejemplo. Supongamos que el usuario sube la imagen de la izquierda de la figura 3.35 y el modelo que está implementado obtiene la segmentación de la derecha de la misma figura.

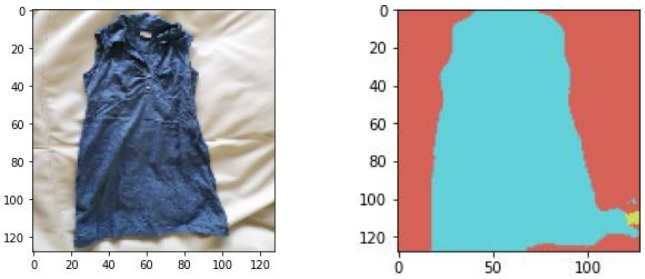


Figura 3.35: Imagen subida y su segmentación.

Como podemos observar, el modelo ha clasificado los pixeles en tres clases: fondo, vestido y camiseta en la parte inferior derecha. Por tanto, aunque para nosotros quede claro que los pixeles de la camiseta están mal clasificados y, por tanto, solo hay una prenda de ropa en la imagen, para el ordenador existen dos prendas de ropa de diferentes clases. Por tanto, si pedimos al sistema que nos recomiende ropa en función de la segmentación, nos recomendará vestidos y camisetas en vez de solo vestidos.

Para evitar este problema, hemos diseñado un criterio para que el sistema pueda decidir el número de prendas de ropa diferentes que aparecen en la imagen. Este criterio se basa en la comparación de los porcentajes de pixeles clasificados en cada una de las clases salvo en la clase fondo.

Supongamos que en la imagen segmentada aparecen los pixeles clasificados en n clases distintas del fondo. Denotamos como $p_1, ..., p_n$ a los porcentajes de pixeles en cada una de las clases y p_{max} al porcentaje máximo.

Entonces, si

$$p_{max} - p_i < 100/n \quad \text{para } i = 1, ..., n, \tag{3.7}$$

consideraremos que la prenda de la clase *i* sí que aparece en la imagen original. En caso contrario, consideraremos que los pixeles clasificados en la clase *i* están mal clasificados.

Por ejemplo, en nuestro caso, los pixeles clasificados como vestido representan el 99,5 % mientras que los clasificados como camiseta solo representan el 0,5 %. Por tanto, según el criterio anterior, dado que $99{,}5 - 0{,}5 \not< 50$ el sistema considera que únicamente el vestido aparece en la imagen original.

Con esta información, el sistema accede a la base de datos que habíamos creado y busca las imágenes que contengan vestidos. De entre todas estas imágenes se seleccionan 6 aleatorias que serán las que se muestren por pantalla. En nuestro caso, las recomendaciones obtenidas han sido las siguientes:

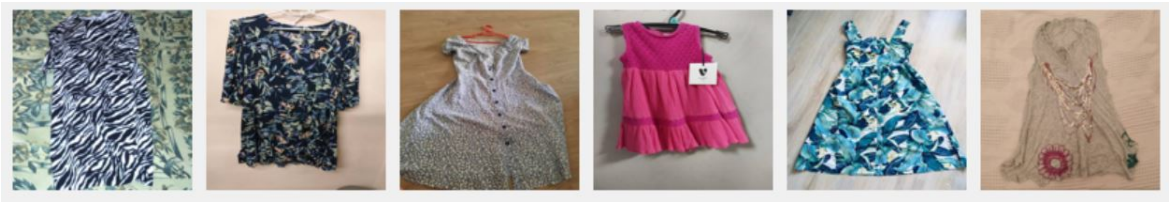


Figura 3.36: Vestidos recomendados.

En el caso de que en la imagen subida existan prendas de distintas clases, el sistema recomendará $\lfloor 6/k \rfloor$ prendas de cada clase donde *k* es el número de clases distintas representadas en la imagen.

3.5.3. Recomendación en función del color

La otra opción de recomendación que ofrece el sistema al usuario es recomendar prendas de ropa similares en función del color de la prenda subida. Para llevar a cabo esta tarea, también vamos a utilizar la segmentación que realiza el modelo de la imagen cargada. Para explicarlo volveremos a usar el ejemplo de la figura 3.35.

Al igual que anteriormente, lo primero que realizamos es ver el número de prendas de diferentes clases que hay en la imagen cargada. En este caso, ya vimos que únicamente había un vestido. Para obtener el color, vamos a recortar la imagen original en función de la posición de los pixeles que se han clasificado como vestido en la imagen segmentada. En nuestro caso, la imagen original recortada en función de la segmentación queda de la siguiente forma:

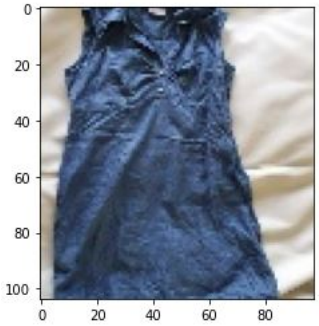


Figura 3.37: Vestido recortado en función de la segmentación.

Con este recorte nos aseguramos que en la imagen predomina la prenda de ropa sobre el fondo y así, podemos aplicar el algoritmo k-medias de la misma manera que lo hicimos en la sección 3.5.1. Repitiendo ese proceso, el color del vestido devuelto por la función es *darkslategray*.

Con la información de que la prenda es un vestido y el color es un gris pizarra oscuro, el sistema accede a la base de datos y busca las imágenes que coincidan con ese par de indicaciones. El problema que vimos es que se podía dar el caso de que no hubiese suficientes imágenes cumpliendo esas dos condiciones. Esto ocurre porque en la mayoría de los casos no hay suficientes imágenes en la base de datos que tengan el mismo color.

Por ello, lo que hicimos para resolver este problema fue introducir una función que, en caso de que no se hubiesen rellenado los seis huecos guardados para la recomendación, el sistema buscase el color más similar al del vestido en el diccionario de colores definido en [14]. En este caso, el color más similar al *darkslategray*. Dado que la similitud entre dos colores puede ser algo subjetivo, decidimos definir una medida de la proporción de similitud entre dos colores de la siguiente manera:

Definición 3.1. Dados dos colores en formato rgb, $c_1 = (r_1, g_1, b_1)$ y $c_2 = (r_2, g_2, b_2)$, si denotamos como:

$$r = \frac{255 - |r_1 - r_2|}{255},$$

$$g = \frac{255 - |g_1 - g_2|}{255},$$

$$b = \frac{255 - |b_1 - b_2|}{255}.$$

Entonces, definimos la proporción de similitud, p, entre esos dos colores como:

$$p = \frac{r + g + b}{3}. \tag{3.8}$$

Notar que p es un valor entre 0 y 1, donde 0 indica que no hay similitud entre los colores y 1 indica que son el mismo color.

Con esta definición, el sistema va comparando el color del vestido con cada uno de los colores definidos en el diccionario mencionado anteriormente. La búsqueda termina cuando se encuentra un color que al menos tenga una proporción de similitud de 0,85. En el caso de que la búsqueda termine sin encontrar ningún color tan similar, la proporción se disminuye en 0,05 por cada vez que se recorre la lista de colores completa.

Tras este proceso obtenemos un nuevo color que usaremos para buscar nuevos vestidos en la base de datos. Si resulta que con este nuevo color no se terminan de llenar los seis huecos de la recomendación se procede a buscar un nuevo color similar a este último.

De esta forma, el proceso se repite hasta que se consigan rellenar los seis huecos guardados para la recomendación. Por ejemplo, en el caso de nuestro vestido, las prendas obtenidas por la recomendación son las siguientes:

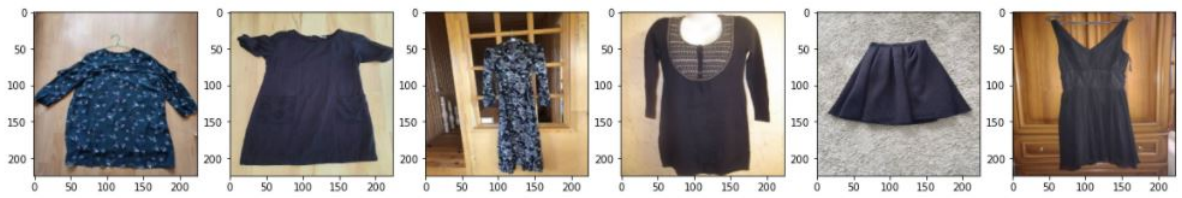


Figura 3.38: Prendas recomendadas en función del color.

En este caso, no había suficientes vestidos de color *darkslategray* por lo que el sistema ha rellenado los huecos con prendas de colores *darkslateblue* y *darkgray* que son los más similares.

Notar además que la quinta imagen es de una falda. Esto es porque, con el objetivo de ampliar el rango de prendas recomendadas, se ha optado por mostrar también prendas similares del mismo color. En el caso de pantalones, vestidos y camisetas se ha ampliado el rango de búsqueda de prendas de la siguiente forma:

- Camisetas: En el caso de que en la imagen cargada al sistema haya una camiseta, el sistema mostrará también polos, camisetas de manga larga y sudaderas.
- Pantalones: En el caso de los pantalones, se mostrarán también chaquetas.
- Vestidos: En el caso de los vestidos, se mostrarán también en las recomendaciones faldas.

Por último, queda por explicar el caso en el que la imagen tenga más de una prenda de ropa. En este caso, todo el proceso de elección del color se realiza para cada prenda de manera independiente. Lo único que cambia es que los huecos de recomendaciones para cada tipo de prenda serán, como en la sección anterior, $\lfloor 6/k \rfloor$ con k el número de clases distintas representadas en la imagen cargada al sistema.

3.5.4. Desarrollo de la aplicación

Durante este capítulo hemos estado comprobando la viabilidad de este proyecto mediante el ajuste y entrenamiento de diferentes modelos y mediante la introducción de dos técnicas de recomendación basadas en los colores de las prendas y en el tipo de prendas. Dado que los resultados obtenidos han sido satisfactorios, para finalizar la fase de viabilidad del proyecto se ha decidido crear una pequeña aplicación que sirva como demo para ver una posible implementación de los diferentes algoritmos y técnicas desarrollados durante este capítulo e intente emular un sistema de recomendación real.

Esta aplicación va a ser realmente una interfaz gráfica de usuario (GUI en inglés) que es un entorno visual donde se permiten añadir botones, texto e imágenes de forma que se permita al usuario interactuar de una forma sencilla con la máquina. Para programarla se utilizará el paquete Tkinter de Python, [31].

Esta aplicación cuenta con un cuadro de texto donde el usuario podrá introducir el directorio de la imagen que quiere subir y también se han añadido cuatro botones con los que el usuario podrá interactuar. El esquema general de la aplicación es el siguiente:

Figura 3.39: Esquema general de la aplicación.

Como se puede ver, en la parte superior hay dos botones. El botón “Borrar” elimina el texto que haya en el cuadro donde se introduce el directorio de la imagen. El segundo, “Cargar imagen” carga la imagen al sistema y permite visualizarla.

Por otro lado, los dos botones inferiores permiten elegir si se quiere una recomendación en función de los colores y el tipo de prenda o únicamente teniendo en cuenta el tipo de prenda de ropa. En particular, el botón rojo permite recomendar prendas del mismo color usando el algoritmo explicado en la sección 3.5.3 y, el botón azul, permite recomendar prendas del mismo tipo utilizando el método desarrollado en la sección 3.5.2.

Para ver en funcionamiento la aplicación vamos a suponer que como usuarios subimos una foto de unos zapatos negros y queremos que nos recomiende prendas similares. El primer paso es introducir el directorio y cargar la imagen.

Figura 3.40: Imagen de los zapatos cargada.

A continuación, vamos a ver qué nos recomienda el sistema si pulsamos ambos botones.

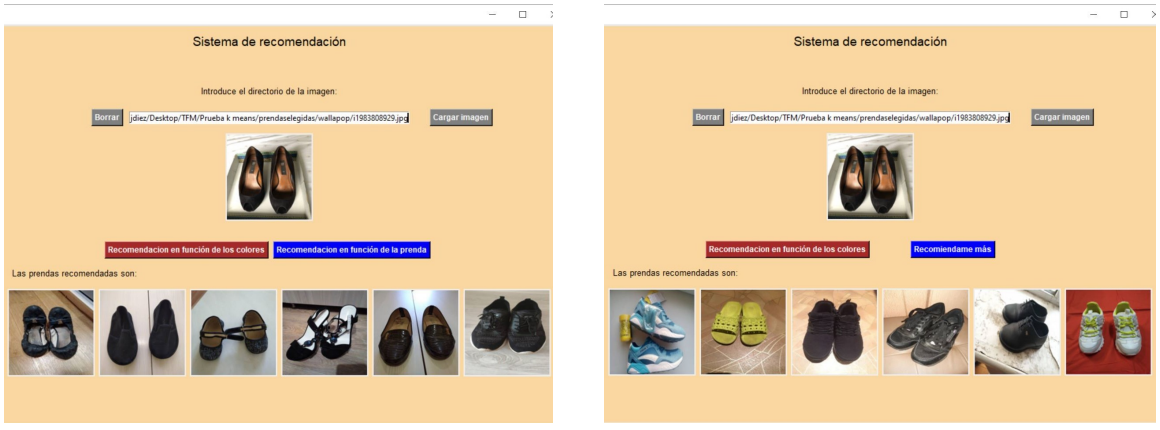


Figura 3.41: Recomendación por colores y prenda a la izquierda y recomendación solo en función de la prenda a la derecha.

Podemos observar las diferencias entre una recomendación y otra. Además, cuando se pulsa el botón de recomendación en función del tipo de prenda de ropa, este cambia de nombre a “Recomiéndame más”, de forma que el usuario puede seguir pulsando para obtener nuevas recomendaciones si las obtenidas no son de su agrado.

Sin embargo, esto no ocurre en la opción de recomendación por colores puesto que en este caso el número de prendas de ropa que hay del mismo color suele ser más limitado y aunque se vuelva a pulsar el botón, la mayoría de veces se recomiendan los mismos productos.

Por último, a esta aplicación se le ha añadido también una parte de gestión de errores que avisará al usuario cuándo el directorio introducido no esté en la forma correcta.

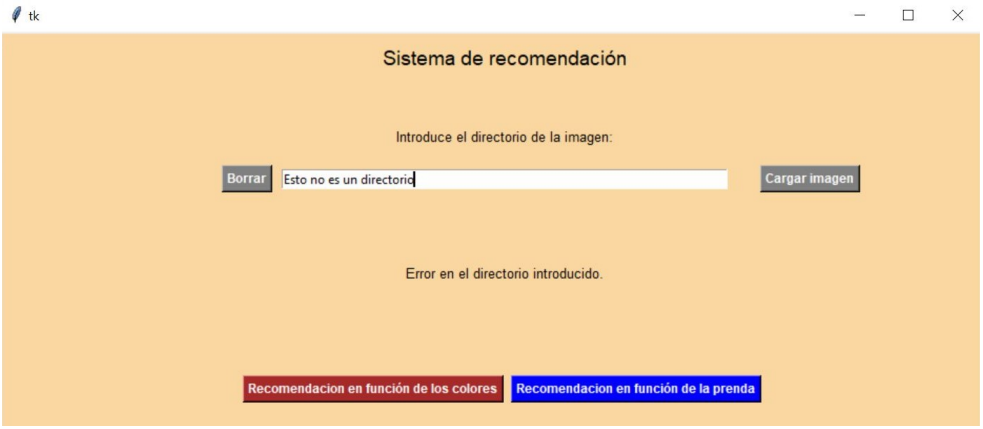


Figura 3.42: Gestión de errores.

3.6. Conclusión

Para concluir, podemos asegurar, por los resultados expuestos en este capítulo, que es viable la construcción de un sistema de recomendación de prendas de ropa utilizando un modelo que incorpora algunas de las técnicas de segmentación semántica expuestas en la parte teórica. De hecho, el rendimiento del último modelo ajustado es bastante bueno, no solo para prendas del conjunto de datos sino también para prendas que aparecen en algunas webs de venta online.

Además, también podemos afirmar que gracias a los algoritmos construidos para el etiquetado de las imágenes y para la creación de la base de datos del sistema de recomendación, se ha conseguido optimizar el tiempo invertido en el proyecto permitiéndonos centrarnos más a fondo en la construcción y optimización de los modelos utilizados.

Por otro lado, queda patente, viendo la aplicación desarrollada, que el uso de técnicas de búsqueda por imagen basados en modelos de segmentación semántica permite, además de mejorar la experiencia del usuario, ofrecer recomendaciones más acertadas que los sistemas tradicionales donde hay que especificar las características principales de la prenda que se sube. De hecho, gracias al uso de la segmentación se ha podido obtener eficazmente el color de las diferentes prendas de ropa que aparecen en las imágenes.

Tras esta fase de viabilidad se pueden plantear las mejoras a realizar antes de entrar en la fase de ejecución del proyecto. Por un lado, como vimos que el modelo final tenía ciertos problemas al intentar segmentar imágenes donde aparecían personas, sería interesante crear un nuevo modelo capaz de detectar las diferentes prendas que tiene una persona. Para ello además de modificar el modelo, habría que cambiar la forma en la que realizamos el etiquetado para el entrenamiento.

Por otro lado, sería interesante refinar el conjunto de datos que tenemos, incluyendo no solo el tipo de prenda de ropa y el color, sino también más características de la prenda como pueden ser si es de

adulto o de niño, si es una prenda deportiva o una prenda de diario, etc. Además, se podrían definir unas nuevas bases de datos donde almacenar toda la información que tenemos actualmente en varias hojas excel con la finalidad de aumentar el volumen de información del que podemos disponer y de ganar en robustez.

En cuanto a la aplicación, se podrían añadir nuevas funcionalidades que permitan añadir más detalle a las recomendaciones como, por ejemplo, la recomendación de prendas complementarias a la subida por el usuario o la recomendación de un kit completo de vestuario. Para ello, habría que definir unas nuevos criterios de recomendación que serían más subjetivos que los mostrados en este trabajo.

Por último, teniendo en cuenta estas consideraciones, se puede pasar a planificar la puesta en producción del modelo en la empresa Efor.

Anexos

Anexo A

Funciones auxiliares

En el siguiente anexo se incluyen las funciones auxiliares que han sido programadas para facilitar la programación del código del proyecto y evitar la presencia de código repetitivo a lo largo de los cuadernos de código. Todas las funciones están programadas en ficheros de *Python* y serán cargadas posteriormente en los cuadernos de *Jupyter*.

Las funciones son las siguientes:

1. Función de preprocesamiento de imágenes.
2. Función de coloreado de la segmentación.
3. Función de etiquetado.
4. Función de construcción de la matriz de confusión.
5. Función para transformar los arrays con las etiquetas y las predicciones del modelo para la posterior construcción de la curva ROC de la clasificación por prenda de ropa.
6. Función de construcción de la curva ROC.
7. Función con las diferentes métricas de validación.
8. Función para buscar el color más similar a otro dado según el criterio definido en este proyecto.
9. Función que devuelve el nombre de un color que está en formato rgb.

Función preprocesamiento de imágenes

redimension.py

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import cv2
import numpy as np

def getImageArr( path , width , height ):
    '''
    Funcion que redimensiona las imagenes originales que vamos a usar para
    entrenar el modelo y la pone en formato de array.

    path: Directorio de la ubicacion de la imagen.

    width: Ancho al que se quiere redimensionar la imagen.

    heigth: Alto al que se quiere redimensionar la imagen.
    '''
    img = cv2.imread(path, 1)
    img = np.float32(cv2.resize(img, ( width , height ))) /127.5 - 1
    return img

def getSegmentationArr( path , nClasses ,  width , height  ):
    '''
    Funcion que permite redimensionar las etiquetas y las transforma en formato
     one-hot encoding.

    path: Directorio de la ubicacion de las etiquetas.

    nClasses: Numero de clases del modelo.

    width: Ancho al que se quiere redimensionar la imagen.

    height: Alto al que se quiere redimensionar la imagen.
    '''

    seg_labels = np.zeros((  height , width  , nClasses ))
    img = cv2.imread(path, 1)
    img = cv2.resize(img, ( width , height ))
    img = img[:, : , 0]

    for c in range(nClasses):
        seg_labels[: , : , c ] = (img == c ).astype(int)
    return seg_labels
```

Función coloreado de la segmentación

colorea_seg.py

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import numpy as np
import seaborn as sns

def colorea_seg(seg,n_classes):
    '''
    Funcion que colorea una imagen de acuerdo a la clase a la que pertenecen
    sus pixeles.

    seg: La imagen que queremos colorear.

    n_classes : Numero de clases del modelo.
    '''

    if len(seg.shape)==3:
        seg = seg[:,:,0]

    seg_img = np.zeros( (seg.shape[0],seg.shape[1],3) ).astype('float')
    colors = sns.color_palette("hls", n_classes)

    for c in range(n_classes):
        segc = (seg == c)
        seg_img[:,:,0] += (segc*( colors[c][0] ))
        seg_img[:,:,1] += (segc*( colors[c][1] ))
        seg_img[:,:,2] += (segc*( colors[c][2] ))

    return(seg_img)
```

Función de etiquetado

etiquetadoultimo.py

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import numpy as np
import csv
import cv2
from sklearn.cluster import KMeans
import os


def etiquetadoultimo(pathcsv,labels, dir_img, dir_seg, input_height,
    input_width):
    '''
    Funcion que genera y guarda las etiquetas de un conjunto de imagenes.

    pathcsv: Directorio donde se encuentra el archivo csv con la informacion
    sobre el conjunto de datos.

    labels: Etiquetas de las distintas clases del modelo.

    dir_img: Directorio de la carpeta donde se encuentran las imagenes que
    queremos etiquetar.

    dir_seg: Directorio de la carpeta donde vamos a guardar las etiquetas.

    input_height: Alto que queremos que tengan las etiquetas.

    input_width: Ancho que queremos que tengan las etiquetas.
    '''
    nombres=[]
    etiquetas=[]

    with open(pathcsv, 'r') as file:
        reader = csv.reader(file, delimiter = ',')
        for row in reader:
            nombres.append(row[0])
            etiquetas.append(row[2])

    images = os.listdir(dir_img)
    for im in (images):
        if(im!='desktop.ini'):
            image=cv2.imread(dir_img + im,1)
            ima= cv2.resize(image,(input_height , input_width))
            s=im.split('.')
            for i in range(len(nombres)):
                if nombres[i].strip() == s[0].strip():
                    etiqueta = etiquetas[i]

            #Aplicacion del primer filtro.

            kernel_1 = np.ones((5,5), np.uint8)
            mascara = cv2.erode( image, kernel_1, iterations = 3)
            mascara = cv2.dilate(mascara, kernel_1, iterations = 1)

            ima= cv2.resize(mascara,(input_height , input_width))
            imac = cv2.cvtColor(ima, cv2.COLOR_BGR2RGB)
            img=imac.reshape((ima.shape[1]*ima.shape[0],3))
```

```
58
59              #Algoritmo k-medias.
60
61              kmeans=KMeans(n_clusters=2, random_state=0)
62              s=kmeans.fit(img)
63              res = np.array(kmeans.labels_).reshape(ima.shape[0], ima.shape[1])
64
65              #Nueva imagen donde a cada pixel se le asocia el valor que le
    corresponde segun la clase a la que pertenezca la prenda de ropa.
66
67              img1 = np.zeros((ima.shape[0],ima.shape[1],3),np.uint8)
68
69              for k in range(len(labels)):
70                  if etiqueta.strip() == labels[k].strip():
71                      ind=k
72
73              for x in range(ima.shape[0]):
74                  for y in range(ima.shape[1]):
75                      if res[x,y]==0:
76                          img1[x,y]=[0,0,0]
77                      if res[x,y]==1:
78                          img1[x,y]=[ind+1,ind+1,ind+1]
79
80              #Aplicacion del segundo filtro.
81
82              img1=cv2.medianBlur(img1, 7)
83
84              #Cambio de mascaras si fuese necesario.
85
86              if(labels[ind].strip()=='T-Shirt' or labels[ind].strip() == 'Dress'
     or labels[ind].strip()== 'Shorts'):
87                  medias=[]
88                  aux=img1.shape[0]//4
89                  paso= (img1.shape[0]-2*aux)//4
90                  for i in range(4):
91                      for j in range(4):
92                          medias.append(np.mean(img1[aux+i*paso:aux+paso+i*paso,
    aux+j*paso:aux+paso+j*paso]))
93
94                  if np.mean(medias)/(ind+1)<0.4:
95                      for x in range(img1.shape[0]):
96                          for y in range(img1.shape[1]):
97                              contador=1
98                              if img1[x,y,1]==0:
99                                  img1[x,y]=[ind+1,ind+1,ind+1]
100                                 contador=10
101
102                             if(contador!=10):
103                                 if img1[x,y,1]==ind+1:
104                                     img1[x,y]=[0,0,0]
105
106             if(labels[ind].strip()=='Pants'):
107                 c= img1[20:50, 40: 80]
108
109                 if np.mean(c)/(ind+1)<0.4:
110                     for x in range(img1.shape[0]):
111                         for y in range(img1.shape[1]):
112                             contador=1
113                             if img1[x,y,1]==0:
114                                 img1[x,y]=[ind+1,ind+1,ind+1]
115                                 contador=10
116
117                             if(contador!=10):
```

```
                                if img1[x,y,1]==ind+1:
                                    img1[x,y]=[0,0,0]


            if(labels[ind].strip()=='Shoes'):
                c=img1[40:80, 40:80]
                if np.mean(c)/(ind+1)<0.4:
                    for x in range(img1.shape[0]):
                        for y in range(img1.shape[1]):
                            contador=1
                            if img1[x,y,1]==0:
                                img1[x,y]=[ind+1,ind+1,ind+1]
                                contador=10

                            if(contador!=10):
                                if img1[x,y,1]==ind+1:
                                    img1[x,y]=[0,0,0]

            #Guardado de la imagen.

            cv2.imwrite(dir_seg + im.replace('.jpg','.png'), img1)
```

Función construcción de la matriz de confusión

confusion_matrix.py

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import numpy as np
import seaborn as sn
import pandas as pd
import matplotlib.pyplot as plt
from tabulate import *
from sklearn.metrics import confusion_matrix

def confusionmatrix(tipo,labels,Yi,y_predi):

    '''
    Funcion que construye la matriz de confusion de la clasificacion por pixel o por prenda segun el tipo que se le pase y guarda la matriz
    como una imagen.

    tipo: Argumento que indica el tipo de matriz de confusion que queremos:

        tabla: Devuelve la matriz de confusion de la clasificacion por pixel en formato de tabla, en este caso sin guardarla.
        colores: Devuelve la matriz de confusion de la clasificacion por pixel normalizada y puesta con colores.
        prenda: Devuelve la matriz de confusion sin normalizar de la clasificacion por prenda de ropa y en el formato de la del tipo
        colores.

    labels: Las etiquetas de las clases que estemos usando en el modelo.

    Yi: Array formado por las etiquetas reales de cada pixel.

    y_predi: Array formado por la prediccion de cada pixel.
    '''

    Nclass = int(np.max(Yi)) + 1
    if(tipo=="tabla"):
        matriz=[[0 for x in range(Nclass)] for y in range(Nclass+1)]
        for c in range(Nclass+1):
            for d in range(Nclass):
                if(c==0):
                    matriz[c][d]= "clase" + str(d)
                else:
                    matriz[c][d]= np.sum((Yi == d)&(y_predi==c-1))

        print(tabulate(matriz, headers='firstrow', showindex=True))

    if(tipo=="colores"):
        matriz=[[0 for x in range(Nclass)] for y in range(Nclass)]
        for c in range(Nclass):
            TP = np.sum( (Yi == c)&(y_predi==c) )
            FN = np.sum( (Yi == c)&(y_predi != c))
            for d in range(Nclass):
                matriz[c][d]= np.sum((Yi == c)&(y_predi==d))/float(TP+FN)

        df_cm = pd.DataFrame(matriz, range(Nclass), range(Nclass))
        fig = plt.figure(figsize=(15,15))
        ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
```

```
        sn.set(font_scale=5)
        cmpix= sn.heatmap(df_cm, annot=True ,annot_kws={"size": 34}, cmap='Blues',cbar=False)
        plt.ylabel('Valores reales')
        plt.xlabel('Valores predichos')
        ax.set_title("Matriz de confusion")
        ax.set_xticks([0.5,1.5,2.5,3.5,4.5,5.5])
        ax.set_xticklabels(labels)
        ax.set_yticks([0.3,1.3,2.3,3.3,4.3,5.3])
        ax.set_yticklabels(labels)

        plt.show()
        cmpix.figure.savefig("cmpix.jpg")

    if(tipo=="prenda"):
        y_testp=[]
        y_predp=[]
        for i in range(len(Yi)):
            valuespred, countspred = np.unique(y_predi[i], return_counts=True)
            valuestrue, countstrue =  np.unique(Yi[i], return_counts=True)
            porcentajes=[]
            for k in range(1,len(valuespred)):
                porcentajes.append(countspred[k]/(np.sum(countspred[1:len(countspred)])))

            aux1=np.max(porcentajes)
            for l in range(len(porcentajes)):
                if(aux1- porcentajes[l]==0):
                    c=valuespred[l+1]

            indmaxtrue= np.max(valuestrue)
            y_testp.append(indmaxtrue)
            y_predp.append(c)


        mc=confusion_matrix(y_testp, y_predp)

        df_cm = pd.DataFrame(mc, labels, labels)
        fig = plt.figure(figsize=(15,15))
        ax = fig.add_axes([0.1, 0.1, 0.8, 0.8])
        sn.set(font_scale=1.4)
        cmp = sn.heatmap(df_cm, annot=True ,annot_kws={"size": 34}, cmap='Blues',cbar=False)
        ax.set_title("Matriz de confusion")
        ax.set_xticks([0.5,1.5,2.5,3.5,4.5])
        ax.set_xticklabels(labels)
        ax.set_yticks([0.3,1.3,2.3,3.3,4.3])
        ax.set_yticklabels(labels)
        plt.ylabel('Valores reales')
        plt.xlabel('Valores predichos')

        plt.show()
        cmp.figure.savefig("cmp.jpg")
```

Función que transforma las etiquetas y predicciones para la curva ROC de la clasificación por prenda

pixel_to_prenda_roc.py

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import numpy as np

def pixel_to_prenda_roc(y_testi, y_predi, y_pred):
    '''
    Funcion que transforma los arrays de etiquetas reales y de predicciones para poder construir la curva ROC
    con los datos de la clasificacion del modelo por prenda de ropa. Devuelve los dos arrays transformados.

    y_testi: Array con las posiciones de los "1" en el formato one-hot encoding de las etiquetas.

    y_predi: Array con las posiciones del elemento de mayor probabilidad en las predicciones.

    y_pred: Array de las predicciones del modelo.
    '''
    n_classes = int(np.max(y_testi)) + 1
    probs=np.empty((y_predi.shape[0], n_classes -1))
    y_testprueb=np.empty((y_testi.shape[0], n_classes-1))
    for i in range(len(y_testi)):
        auxt=[]
        valuespred, countspred = np.unique(y_predi[i], return_counts=True)
        valuestrue, countstrue =  np.unique(y_testi[i], return_counts=True)
        porcentajes=[]
        for k in range(1,len(valuespred)):
            porcentajes.append(countspred[k]/(np.sum(countspred[1:len(countspred)])))

        aux1=np.max(porcentajes)
        for l in range(len(porcentajes)):
            if(aux1- porcentajes[l]==0):
                c = valuespred[l+1]

        true_points = np.argwhere(y_predi[i]==c)
        v=list()
        medias=[]
        for b in range(len(true_points)):
            v.append(y_pred[i,true_points[b,0]-true_points.min(axis=0)[0], true_points[b,1]-true_points.min(axis=0)[1]])

        v1=np.array(v)
        media=v1.mean(axis=0)
        mediar = media[1:n_classes]
        for j in range(n_classes-1):
            mediar[j] = mediar[j] + media[0]/(n_classes-1)
        medias.append(mediar)

        indmaxtrue= np.max(valuestrue)

        for j in range(n_classes):
            if(indmaxtrue==j):
```

```
                auxt.append(1)

            else:
                auxt.append(0)

        auxt.pop(0)
        y_testprueb[i,:]= auxt
        probs[i,:]=list(mediar)

    return y_testprueb, probs
```

Función construcción de la curva ROC

Curva_ROC.py

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import numpy as np
import matplotlib.pyplot as plt
from itertools import cycle
from sklearn.metrics import roc_curve, auc

def Curva_ROC(tipo,labels, y_test, y_pred):
    '''
    Funcion que dibuja las curvas ROC del problema multiclase y realiza un zoom en la esquina izquierda.
    tipo: Si es "pixel" se dibujara la curva ROC de la clasificacion del modelo por pixel y si es "prendas" se dibujara
    la curva ROC de la clasificacion del modelo por prenda de ropa.

    labels: etiquetas de las clases utilizadas en el modelo.

    y_test: Array formado por las imagenes con las etiquetas verdaderas.

    y_pred: Array formado por las predicciones del modelo.
    '''
    lw = 2

    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    if(tipo=='pixel'):
        n_classes = y_test.shape[3]
    if(tipo=='prendas'):
        n_classes = 5
    for i in range(n_classes):
        if(tipo=='pixel'):
            fpr[i], tpr[i], _ = roc_curve(y_test[:,:,:, i].ravel(), y_pred[:,:,:, i].ravel())
        if(tipo=='prendas'):
            fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_pred[:, i])

        roc_auc[i] = auc(fpr[i], tpr[i])



    # Curvas ROC
    plt.figure(1,figsize=(10,7))

    colores = cycle(['aqua', 'darkorange', 'cornflowerblue', 'red', 'green', 'purple'])
    for i, color in zip(range(n_classes), colores):
        plt.plot(fpr[i], tpr[i], color=color, lw=lw,
                 label='clase {} vs. resto (area = {:0.2f})'
                 ''.format(labels[i], roc_auc[i]))

    plt.plot([0, 1], [0, 1], 'k--', lw=lw)
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('Tasa de falsos positivos')
    plt.ylabel('Tasa de verdaderos positivos')
```

```
    if(tipo=='pixel'):
        plt.title('Curvas ROC de las 6 clases del modelo')
    if(tipo=='prendas'):
        plt.title('Curvas ROC de las 5 prendas del modelo')

    plt.legend(loc="lower right")
    plt.show()

    plt.figure(2,figsize=(10,7))
    plt.xlim(0, 0.2)
    plt.ylim(0.8, 1)

    colores = cycle(['aqua', 'darkorange', 'cornflowerblue', 'red', 'green', '
    purple'])
    for i, color in zip(range(n_classes), colores):
        plt.plot(fpr[i], tpr[i], color=color, lw=lw,
                 label='clase {} vs. resto (area = {:0.2f})'
                 ''.format(labels[i], roc_auc[i]))

    plt.plot([0, 1], [0, 1], 'k--', lw=lw)
    plt.xlabel('Tasa de falsos positivos')
    plt.ylabel('Tasa de verdaderos positivos')
    plt.title('Zoom en la esquina izquierda')
    plt.show()
```

Función con las métricas de validación

metrics.py

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


import numpy as np
import keras.backend as K

def metrics(tipo,labels, Yi,y_predi):
    '''
    Funcion que calcula la metrica que se le pasa como parametro para validar la segmentacion del modelo.

    tipo: El tipo de medida de validacion que se quiere calcular. Puede ser de los siguientes tipos:

        MPA: Obtiene la metrica de precision por pixel en cada clase y devuelve tambien la media de todos los valores.
        MIoU: Obtiene la metrica IoU por clase y devuelve tambien la media de todos los valores.
        MDice: Obtiene el coeficiente Dice por clase y devuelve tambien la media de todos los valores.
        FWIoU: Calcula una version ponderada del MIoU llamada "Frequency Weighted Intersection over Union".
        prenda_acc: Obtiene la precision de la clasificacion por prenda de ropa del modelo.
        loc_acc: Obtiene la precision obtenida a la hora de localizar la prenda de ropa.

    labels: Las etiquetas de las clases usadas en el modelo.

    Yi: Array formado por las etiquetas reales de cada pixel.

    y_predi: Array formado por la prediccion de cada pixel.
    '''

    PAs=[]
    IoUs = []
    FWIous=[]
    Dices=[]
    FPs=[]
    Nclass = int(np.max(Yi)) + 1
    for c in range(Nclass):
        TP = np.sum( (Yi == c)&(y_predi==c) ) #Verdaderos positivos
        FP = np.sum( (Yi != c)&(y_predi==c) ) #Falsos positivos
        FN = np.sum( (Yi == c)&(y_predi != c)) #Falsos negativos
        TN = np.sum( (Yi != c)&(y_predi != c)) #Verdaderos negativos
        if(tipo=="MPA"):
            PA=float(TP+TN)/float(TP+TN+FP+FN)
            PAs.append(PA)
            print("clase {}: #TP={:6.0f}, #FP={:6.0f}, #FN={:5.0f}, #TN={:5.0f}, PA={:4.3f}".format(labels[c],TP,FP,FN,TN,PA))
            if(c==Nclass-1):
                MPA= np.mean(PAs)
                print("____________________________")
                print("Mean Pixel Accuracy: {:4.3f}".format(MPA))

        if(tipo=="MIoU"):
            IoU = TP/float(TP + FP + FN)
```

```
52              print("clase {}: #TP={:6.0f}, #FP={:6.0f}, #FN={:5.0f}, IoU={:4.3f}
    ".format(labels[c],TP,FP,FN,IoU))
53              IoUs.append(IoU)
54              if(c==Nclass-1):
55                  mIoU = np.mean(IoUs)
56                  print("_________________")
57                  print("Mean IoU: {:4.3f}".format(mIoU))
58
59          if(tipo=="MDice"):
60              Dice= 2*TP/float(2*TP + FP + FN)
61              Dices.append(Dice)
62              print("clase {}: #TP={:6.0f}, #FP={:6.0f}, #FN={:5.0f}, Dice={:4.3f
    }".format(labels[c],TP,FP,FN,Dice))
63              if(c==Nclass-1):
64                  MDice = np.mean(Dices)
65                  print("Dice coefficient: {:4.3f}".format(MDice))
66
67          if(tipo=="FWIou"):
68              FWIouaux = (float(TP)*float(TP+FP))/float(TP + FP + FN)
69              FWIous.append(FWIouaux)
70              FPs.append(float(FP)+float(TP))
71              if(c==Nclass-1):
72                  FWIou = np.sum(FWIous)/np.sum(FPs)
73                  print("Frequency Weighted Intersection over Union: {:4.3f}".
    format(FWIou))
74
75      if(tipo=="prenda_acc"):
76          buenos=0
77          for i in range(Yi.shape[0]):
78              valuespred, countspred = np.unique(y_predi[i], return_counts=True)
79              valuestrue, countstrue =  np.unique(Yi[i], return_counts=True)
80              porcentajes=[]
81              for k in range(1,len(valuespred)):
82                  porcentajes.append(countspred[k]/(np.sum(countspred[1:len(
    countspred)])))
83
84              aux1=np.max(porcentajes)
85              indices=[]
86              for l in range(len(porcentajes)):
87                  if(aux1- porcentajes[l]==0):
88                      k = valuespred[l+1]
89
90              indmaxtrue= np.max(valuestrue)
91              if k==indmaxtrue:
92                  buenos = buenos+1
93
94          accuracy = (buenos/Yi.shape[0])*100
95          print("Precision de acierto en la prenda de ropa: {:4.3f}".format(
    accuracy) + "%")
96
97      if(tipo=="loc_acc"):
98          s = y_predi-Yi
99          loc_acc = (np.sum(s==0)/(np.sum(s!=0)+np.sum(s==0)))*100
100         print("Precision en la localizacion del objeto: {:4.3f}".format(loc_acc
    ) + "%")
```

Función búsqueda de color más similar

encontrarcolor.py

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


from webcolors import (
    CSS3_NAMES_TO_HEX,
    hex_to_rgb,
)

def hexColorDelta(hex1, hex2):
    '''
    Funcion que devuelve la proporcion de similitud entre dos colores pasados en formato hexadecimal.

    hex1: Primer color en forma hexadecimal.

    hex2: Segundo color en forma hexadecimal.
    '''

    col1 = hex_to_rgb(hex1)
    col2 = hex_to_rgb(hex2)

    r1 = col1[0];
    g1 = col1[1];
    b1 = col1[2];

    r2 = col2[0];
    g2 = col2[1];
    b2 = col2[2];

    r = 255 - abs(r1 - r2);
    g = 255 - abs(g1 - g2);
    b = 255 - abs(b1 - b2);

    r /= 255;
    g /= 255;
    b /= 255;

    return (r + g + b) / 3;

def encontrarcolor(color,lista):
    '''
    Funcion que devuelve el color mas similar de los que hay en una lista de acuerdo a la proporcion de la funcion anterior.

    color: Color del cual se quiere encontrar el color mas parecido.

    lista: Lista de colores de los que se va a sacar el color mas parecido.
    '''
    css3_db = CSS3_NAMES_TO_HEX
    i=0
    r=0.85
    encontrado=False
    while encontrado==False:
        p = hexColorDelta(css3_db[color], lista[i])
        if (p>r and p<1):
            encontrado=True
            return(lista[i])
```

```
59
60          i=i+1
61          if(i==len(lista)):
62               i=0
63               r  =  r-0.05
```

Función rgb a nombre del color

rgb_to_nombre.py

```python
#!/usr/bin/env python
# coding: utf-8

# In[4]:


from scipy.spatial import KDTree
from webcolors import (
    CSS3_NAMES_TO_HEX,
    hex_to_rgb,
)
def rgb_to_nombre(rgb_tuple):
    '''
    Funcion que busca el nombre del color mas cercano al que se pasa como argumento en forma rgb
    en un diccionario de colores definido previamente.

    rgb_tuple: Tupla con los valores de rojo verde y azul que forman el color del que queremos obtener el nombre.
    '''
    css3_dict = CSS3_NAMES_TO_HEX
    nombres = []
    valores_rgb = []
    for color_hex, nombre in css3_dict.items():
        nombres.append(color_hex)
        valores_rgb.append(hex_to_rgb(nombre))

    kdt_db = KDTree(valores_rgb)
    distancia, indice = kdt_db.query(rgb_tuple)
    return nombres[indice]
```

Anexo B

Código modelo final

En este segundo anexo se incluye el cuaderno de *Jupyter* que incluye todo el proceso de creación y análisis del modelo final. En particular, en el cuaderno se incluye lo siguiente:

- Carga del conjunto de datos y etiquetado de las imágenes.
- Programación de la arquitectura de la red, división del conjunto de datos en entrenamiento y validación y entrenamiento de la red.
- Validación del modelo con la visualización de las predicciones realizadas por el modelo en algunas imágenes del conjunto de validación, con la construcción de las matrices de confusión y de las curvas ROC de la clasificación a nivel de pixel y de la clasificación por prenda de ropa y con la evaluación de algunas métricas de validación.

Modelo final

Carga de todos los paquetes y funciones necesarias.

[1]:
```
%cd C:\Users\jdiez\Desktop\TFM\Prueba k means
%run redimension.py
%run colorea_seg.py
%run etiquetado.py
%run metrics.py
%run confusionmatrix.py
%run CurvaROC.py
import matplotlib.pyplot as plt
import os
import tensorflow as tf
from tensorflow import keras
from keras import initializers
from keras.layers import Conv2D, MaxPooling2D, Conv2DTranspose, concatenate
from keras.layers import Activation, Dropout, Input
from tensorflow import random
from keras.models import *
from numpy.random import seed
import csv
from sklearn.utils import shuffle
```

Definición de los directorios, de las clases del modelo y de los tamaño de entrada y salida de las imágenes.

[2]:
```
dir_data = "prendaselegidas/"
dir_seg = dir_data + "labels/"
dir_img = dir_data + "imagenes/"
```

[3]:
```
labels = ['T-Shirt','Pants', 'Dress', 'Shorts', 'Shoes']
```

[4]:
```
input_height=128
input_width=128
output_height , output_width = 128 , 128
input_shape=(input_height,input_width, 3)
n_classes=len(labels)+1
```

Etiquetado

[5]:
```
etiquetadoultimo('images.csv',labels, dir_img, dir_seg, input_height,␣
↪input_width)
```

Arquitectura del modelo

[6]:
```
np.random.seed(12345)
tf.random.set_seed(12345)

init=initializers.glorot_uniform(seed=2)

img_input = Input(shape=(input_height,input_width, 3))

# Fase de codificacion

conv1 = Conv2D(16, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(img_input)
conv1 = Conv2D(16, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(conv1)
pool1 = MaxPooling2D((2, 2))(conv1)

conv2 = Conv2D(32, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(pool1)
conv2 = Conv2D(32, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(conv2)
pool2 = MaxPooling2D((2, 2))(conv2)

conv3 = Conv2D(64, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(pool2)
conv3 = Conv2D(64, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(conv3)
pool3 = MaxPooling2D((2, 2))(conv3)

conv4 = Conv2D(128, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(pool3)
conv4 = Conv2D(128, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(conv4)
pool4 = MaxPooling2D((2, 2))(conv4)

conv5 = Conv2D(256, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(pool4)
conv5 = Conv2D(256, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(pool4)

# Fase de decodificacion

convt6 = Conv2DTranspose( 256 , kernel_size=(2,2) ,  strides=(2,2) ,␣
 ↪padding='same', kernel_initializer=init)(conv5)
convt6 = concatenate([convt6, conv4])
```

```
conv6 = Conv2D(128, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(convt6)
conv6 = Conv2D(128, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(conv6)

convt7 = Conv2DTranspose( 128 , kernel_size=(2,2) ,  strides=(2,2) ,␣
 ↪padding='same', kernel_initializer=init)(conv6)
convt7 = concatenate([convt7, conv3])

conv7 = Conv2D(64, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(convt7)
conv7 = Conv2D(64, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(conv7)

convt8 = Conv2DTranspose( 64 , kernel_size=(2,2) ,  strides=(2,2) ,␣
 ↪padding='same')(conv7)
convt8 = concatenate([convt8, conv2])

conv8 = Conv2D(32, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(convt8)
conv8 = Conv2D(32, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(conv8)


convt9 = Conv2DTranspose( 32 , kernel_size=(2,2) ,  strides=(2,2) ,␣
 ↪padding='same')(conv8)
convt9 = concatenate([convt9, conv1])

conv9 = Conv2D(16, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(convt9)
conv9 = Conv2D(16, (3, 3), activation='elu', padding='same',␣
 ↪kernel_initializer=init)(conv9)

#Capa de salida
out= Conv2D(n_classes, (1, 1), activation='softmax')(conv9)

model = Model(img_input, out)
model.summary()
```

```
Model: "model"
_________________________________________________________________________________
Layer (type)
=================================================================================
Total params: 1,721,270
Trainable params: 1,721,270
Non-trainable params: 0
_________________________________________________________________________________
```

Definición de la función de pérdida y optimizador que se van a utilizar en el entrenamiento.

[7]:
```
model.compile(loss='categorical_crossentropy',
              optimizer= 'adam',
              metrics=['accuracy'])
```

Creación de conjuntos de entrenamiento y validación

[8]:
```
images = os.listdir(dir_img)
images.remove('desktop.ini')
images.sort()
segmentations  = os.listdir(dir_seg)
segmentations.remove('desktop.ini')
segmentations.sort()
```

[9]:
```
X = []
Y = []
for im , seg in zip(images,segmentations) :
        X.append( getImageArr(dir_img + im , input_width , input_height )  )
        Y.append( getSegmentationArr( dir_seg + seg , n_classes , output_width ,␣
 ↪output_height )  )

X, Y = np.array(X) , np.array(Y)
```

División en entrenamiento y validación.

[10]:
```
np.random.seed(12345)
train_rate = 0.8
index_train = np.random.choice(X.shape[0],int(X.
 ↪shape[0]*train_rate),replace=False)
index_test  = list(set(range(X.shape[0])) - set(index_train))


X, Y = shuffle(X,Y)
X_train, y_train = X[index_train],Y[index_train]
X_test, y_test = X[index_test],Y[index_test]
```

Entrenamiento del modelo con 25 épocas.

[11]:
```
batch_size=32
hist=model.fit(X_train,y_train,
          steps_per_epoch = len(X_train) // batch_size,
                  validation_data=(X_test,y_test),
          validation_steps = len(X_test)//batch_size
                 ,epochs=25)
```

```
Epoch 1/25
25/25 [==============================] - 137s 4s/step - loss: 1.5596 - accuracy:
0.4061 - val_loss: 1.3388 - val_accuracy: 0.5328
```

```
Epoch 2/25
25/25 [==============================] - 87s 4s/step - loss: 1.3061 - accuracy:
0.5133 - val_loss: 1.2402 - val_accuracy: 0.5377
Epoch 3/25
25/25 [==============================] - 88s 4s/step - loss: 1.2035 - accuracy:
0.5329 - val_loss: 1.1427 - val_accuracy: 0.5661
Epoch 4/25
25/25 [==============================] - 88s 4s/step - loss: 1.1376 - accuracy:
0.5630 - val_loss: 1.1079 - val_accuracy: 0.5683
Epoch 5/25
25/25 [==============================] - 88s 4s/step - loss: 1.1263 - accuracy:
0.5638 - val_loss: 1.0674 - val_accuracy: 0.5829
Epoch 6/25
25/25 [==============================] - 87s 4s/step - loss: 1.0773 - accuracy:
0.5794 - val_loss: 1.0332 - val_accuracy: 0.6206
Epoch 7/25
25/25 [==============================] - 88s 4s/step - loss: 1.0159 - accuracy:
0.6190 - val_loss: 0.9507 - val_accuracy: 0.6515
Epoch 8/25
25/25 [==============================] - 87s 3s/step - loss: 0.9831 - accuracy:
0.6412 - val_loss: 0.8936 - val_accuracy: 0.6958
Epoch 9/25
25/25 [==============================] - 88s 4s/step - loss: 0.8242 - accuracy:
0.7068 - val_loss: 0.8178 - val_accuracy: 0.7058
Epoch 10/25
25/25 [==============================] - 87s 3s/step - loss: 0.8112 - accuracy:
0.7056 - val_loss: 0.7369 - val_accuracy: 0.7376
Epoch 11/25
25/25 [==============================] - 87s 4s/step - loss: 0.7473 - accuracy:
0.7397 - val_loss: 0.6686 - val_accuracy: 0.7795
Epoch 12/25
25/25 [==============================] - 87s 3s/step - loss: 0.6053 - accuracy:
0.8004 - val_loss: 0.6349 - val_accuracy: 0.7870
Epoch 13/25
25/25 [==============================] - 89s 4s/step - loss: 0.5535 - accuracy:
0.8166 - val_loss: 0.5860 - val_accuracy: 0.8183
Epoch 14/25
25/25 [==============================] - 89s 4s/step - loss: 0.4848 - accuracy:
0.8395 - val_loss: 0.5571 - val_accuracy: 0.8152
Epoch 15/25
25/25 [==============================] - 88s 4s/step - loss: 0.4458 - accuracy:
0.8563 - val_loss: 0.5529 - val_accuracy: 0.8229
Epoch 16/25
25/25 [==============================] - 87s 3s/step - loss: 0.4169 - accuracy:
0.8664 - val_loss: 0.5094 - val_accuracy: 0.8418
Epoch 17/25
25/25 [==============================] - 88s 4s/step - loss: 0.3783 - accuracy:
0.8783 - val_loss: 0.5042 - val_accuracy: 0.8451
```

```
Epoch 18/25
25/25 [==============================] - 87s 3s/step - loss: 0.2995 - accuracy:
0.9044 - val_loss: 0.5724 - val_accuracy: 0.8412
Epoch 19/25
25/25 [==============================] - 85s 3s/step - loss: 0.3508 - accuracy:
0.8887 - val_loss: 0.5119 - val_accuracy: 0.8429
Epoch 20/25
25/25 [==============================] - 88s 4s/step - loss: 0.2969 - accuracy:
0.9060 - val_loss: 0.5083 - val_accuracy: 0.8543
Epoch 21/25
25/25 [==============================] - 87s 3s/step - loss: 0.2476 - accuracy:
0.9207 - val_loss: 0.5246 - val_accuracy: 0.8534
Epoch 22/25
25/25 [==============================] - 86s 3s/step - loss: 0.2354 - accuracy:
0.9239 - val_loss: 0.4763 - val_accuracy: 0.8677
Epoch 23/25
25/25 [==============================] - 87s 3s/step - loss: 0.1801 - accuracy:
0.9420 - val_loss: 0.4791 - val_accuracy: 0.8721
Epoch 24/25
25/25 [==============================] - 86s 3s/step - loss: 0.1776 - accuracy:
0.9423 - val_loss: 0.5092 - val_accuracy: 0.8640
Epoch 25/25
25/25 [==============================] - 87s 4s/step - loss: 0.1694 - accuracy:
0.9437 - val_loss: 0.5281 - val_accuracy: 0.8704
```

Dibujo de las gráficas de evolución de las funciones de pérdida.

[12]:
```
contador=0
for key in ['loss', 'val_loss']:
    contador=contador+1
    if contador==1:
        plt.plot(hist.history[key],marker='.',linestyle='-',label='Función de␣
 ↪perdida entrenamiento')
    if(contador==2):
        plt.plot(hist.history[key],marker='.',linestyle='-',label='Función de␣
 ↪perdida validación')

plt.title('Funciones de perdida en el conjunto de entrenamiento y validación')
plt.xlabel('Épocas')
plt.ylabel('Función de pérdida')
plt.legend(loc='best', prop={'size': 13})
plt.grid(False)
plt.show()
```

Dibujo de las gráficas de evolución de la precisión.

[13]:
```
contador=0
for key in ['accuracy', 'val_accuracy']:
    contador=contador+1
    if contador==1:
        plt.plot(hist.history[key],marker='.',linestyle='-',label='Precisión␣
 ↪conjunto de entrenamiento')
    if(contador==2):
        plt.plot(hist.history[key],marker='.',linestyle='-',label='Precisión␣
 ↪conjunto de validación')

plt.title('Precisión en el conjunto de entrenamiento y validación')
plt.legend(loc='best', prop={'size': 12})
plt.xlabel('Épocas')
plt.ylabel('Precisión')
plt.grid(False)
plt.show()
```

Guardado del modelo.

[14]:
```
model.save('modelofinal.h5')
```

Validación del modelo

Carga del modelo.

[15]:
```
model = tf.keras.models.load_model('modelofinal.h5')
```

Predicciones en el conjunto de validación

[16]:
```
y_pred = model.predict(X_test)
y_predi = np.argmax(y_pred, axis=3)
y_testi = np.argmax(y_test, axis=3)
```

[17]:
```
for i in range(10):
    img_is  = (X_test[i] + 1)*(255.0/2)
    img_is = cv2.cvtColor(img_is, cv2.COLOR_BGR2RGB)
    seg = y_predi[i]
    segtest = y_testi[i]

    fig = plt.figure(figsize=(10,30))
    ax = fig.add_subplot(1,3,1)
    ax.imshow(img_is/255.0)
    ax.grid(False)
    ax.set_title("original")

    ax = fig.add_subplot(1,3,2)
```

```
    ax.imshow(coloreaseg(seg,n_classes))
    ax.grid(False)
    ax.set_title("Clase predicha")

    ax = fig.add_subplot(1,3,3)
    ax.imshow(coloreaseg(segtest,n_classes))
    ax.grid(False)
    ax.set_title("Clase verdadera")
    plt.show()
```

Predicciones en un conjunto test

[18]:
```
dir_test1 = dir_data + "wallapop/"
```

[19]:
```
images4 = os.listdir(dir_test1)
images4.sort()

Z1 = []
for im in (images4) :
    Z1.append( getImageArr(dir_test1 + im , input_width , input_height )  )

Z1 = np.array(Z1)
```

[20]:
```
y_pred2 = model.predict(Z1)
y_predi2 = np.argmax(y_pred2, axis=3)
```

[21]:
```
for i in range(10):
    img_is  = (Z1[i] + 1)*(255.0/2)
    seg = y_predi2[i]
    imac = cv2.cvtColor(img_is, cv2.COLOR_BGR2RGB)

    fig = plt.figure(figsize=(10,30))
    ax = fig.add_subplot(1,2,1)
    ax.imshow(imac/255.0)
    ax.grid(False)
    ax.set_title("original")

    ax = fig.add_subplot(1,2,2)
    ax.imshow(coloreaseg(seg,n_classes))
    ax.grid(False)
    ax.set_title("Clase predicha")
```

Otras técnicas de validación

Matrices de confusión.

Definición del nombre de las clases para las matrices de confusión.

[22]:
```
labelsamp = ['Fondo','Camiseta','Pantalones', 'Vestido', 'Shorts', 'Zapatos']
```

[23]:
```
labelst= ['Camiseta','Pantalones', 'Vestido', 'Shorts', 'Zapatos']
```

Matriz de confusión de la clasificación a nivel de pixel.

[24]:
```
confusionmatrix('colores', labelsamp, y_testi,y_predi)
```

Matriz de confusión de la clasificación por tipo de prenda de ropa.

[25]:
```
confusionmatrix('prenda', labelst, y_testi,y_predi)
```

Curvas ROC.

Transformación de los arrays que contienen las etiquetas de las imágenes del conjunto de validación y las predicciones del modelo para realizar el gráfico de la curva ROC de la clasificación por tipo de prenda de ropa.

[26]:
```
y_testprueb, probs= pixel_to_prenda_roc(y_testi, y_predi, y_pred)
```

Dibujo de la curva ROC de la clasificación por tipo de prenda de ropa.

[27]:
```
Curva_ROC('prendas',labelst,y_testprueb, probs)
```

Dibujo de la curva ROC de la clasificación a nivel de pixel.

[28]:
```
Curva_ROC('pixel',labelsamp, y_test, y_pred)
```

Métricas de validación.

Cálculo de la métrica MIoU y de la métrica de precisión de la clasificación por tipo de prenda de ropa.

[29]:
```
metrics('MIoU',labelsamp, y_testi, y_predi)
```

```
clase Fondo: #TP=1537481, #FP=106097, #FN=90795, IoU=0.886
clase T-Shirt: #TP=248308, #FP= 66040, #FN=49281, IoU=0.683
clase Pants: #TP=253749, #FP= 40604, #FN=63716, IoU=0.709
clase Dress: #TP=325735, #FP= 71042, #FN=93646, IoU=0.664
clase Shorts: #TP=201006, #FP=106667, #FN=60761, IoU=0.546
clase Shoes: #TP=285861, #FP= 34210, #FN=66461, IoU=0.740
------------------
Mean IoU: 0.705
```

[30]:
```
metrics('prenda_acc',  y_testi, y_predi)
```

```
Precisión de acierto en la prenda de ropa: 90.000%
```

Anexo C

Aplicación

En este último anexo se incluye el cuaderno de *Python* que contiene el código de la interfaz gráfica de usuario donde se incluye el sistema de recomendación. Dentro del cuaderno se encuentran las siguientes cuatro funciones:

- **Función cargaimagen**: Función que permite cargar y visualizar la imagen que se pasa como directorio.
- **Función recomendacion**: Función que recomienda ropa en función del tipo de prenda y de los colores que tenga la imagen subida por el usuario.
- **Función recomendacionprenda**: Función que recomienda ropa teniendo en cuenta únicamente el tipo de prenda que aparece en la imagen subida por el usuario.
- **Función clear**: Función que borra el contenido del panel donde se ha escrito el directorio.

Interfaz gráfica de usuario

Aplicacion.py

```
#!/usr/bin/env python
# coding: utf-8

# In[]:


from encontrarcolor import*
from rgb_to_nombre import*
from redimension import*
import tkinter as tk
from PIL import Image, ImageTk
import numpy as np
import csv
import cv2
import math
from sklearn.cluster import KMeans
from numpy.random import seed
import tensorflow as tf
from webcolors import (
    CSS3_NAMES_TO_HEX,
    hex_to_rgb,
)
import matplotlib.pyplot as plt
import random

root= tk.Tk() #Ventana raiz de la aplicacion.

canvas1 = tk.Canvas(root, width = 930, height = 600,  relief = 'raised', bg = '#fad7a0') #Ventana donde se van a incluir todos los elementos.
canvas1.pack()

label1 = tk.Label(root, text='Sistema de recomendacion',bg = '#fad7a0') #Titulo de la aplicacion.
label1.config(font=('helvetica', 14))
canvas1.create_window(480, 25, window=label1)

label2 = tk.Label(root, text='Introduce el directorio de la imagen:', bg = '#fad7a0')
label2.config(font=('helvetica', 10))
canvas1.create_window(480, 100, window=label2)

entry1 = tk.Entry (root, width = 70) #Lugar donde el usuario tiene que introducir el directorio.
canvas1.create_window(480, 140, window=entry1)

def cargaimagen():
    '''
    Funcion que permite cargar y visualizar la imagen que se introduce como directorio.
    '''

    dir_img = entry1.get()
    try:
        imagens=Image.open(dir_img)
        imagens= imagens.resize((128,128))
        test1 = ImageTk.PhotoImage(imagens)
        label3 = tk.Label(image=test1)
        label3.image = test1
        canvas1.create_window(480, 230, window=label3)

```

```
    except Exception:
        label4 = tk.Label(root, text= 'Error en el directorio introducido.', font=('helvetica', 10), bg = '#fad7a0')
        canvas1.create_window(480, 230, window=label4)

css3_db = CSS3_NAMES_TO_HEX

def recomendacion():
    '''
    Funcion que recomienda ropa en funcion del tipo y de los colores de la prenda subida por el usuario. Esta funcion permite visualizar las seis recomendaciones.
    '''
    dir_img = entry1.get()
    label6 = tk.Label(root, text= 'Las prendas recomendadas son:',font=('helvetica', 10), bg = '#fad7a0')
    canvas1.create_window(185, 375, window=label6)
    truelabels = ['Fondo','T-Shirt','Pants', 'Dress', 'Shorts', 'Shoes']
    height =128
    width = 128

    model = tf.keras.models.load_model('modelofinal.h5')
    Z=[]
    imagen=getImageArr( dir_img , width , height )
    Z.append(imagen)
    Z=np.array(Z)

    y_pred = model.predict(Z)
    y_predi = np.argmax(y_pred, axis=3)

    aux=y_predi[0].shape[0]//10
    recort=y_predi[0][aux:y_predi[0].shape[0]-aux,aux:y_predi[0].shape[0]-aux]
    values, counts = np.unique(recort, return_counts=True)
    porcentajes=[]
    for k in range(1,len(values)):
        porcentajes.append(counts[k]/(np.sum(counts[1:len(counts)])))

    aux1=np.max(porcentajes)
    indices=[]
    for l in range(len(porcentajes)):
        if(aux1- porcentajes[l]<(1/len(porcentajes))):
            indices.append(values[l+1])

    coloresprimeros =[]
    for i in indices:
        true_points = np.argwhere(recort==i)
        top_left = true_points.min(axis=0)
        bottom_right = true_points.max(axis=0)
        crop = (imagen[aux:y_predi[0].shape[0]-aux,aux:y_predi[0].shape[1]-aux]+1)[top_left[0]:bottom_right[0]+1,top_left[1]:bottom_right[1]+1]
        crop = crop*(255.0/2)
        crop = cv2.cvtColor(crop, cv2.COLOR_BGR2RGB)
        img=crop.reshape((crop.shape[1]*crop.shape[0],3))

        kmeans=KMeans(n_clusters=4, random_state=0)
        s=kmeans.fit(img)
        labels=kmeans.labels_
        centroid=kmeans.cluster_centers_
        labels=list(labels)
        m=np.array(labels).reshape(crop.shape[0], crop.shape[1])

        v=list()
        for i in range(len(true_points)):
```

```
            v.append(m[true_points[i,0]-true_points.min(axis=0)[0],true_points[
    i,1]-true_points.min(axis=0)[1]])

        percent=[]
        for k in range(len(centroid)):
            j=v.count(k)
            j=j/(len(v))
            percent.append(j)

        coloresordenados =np.zeros((len(percent), centroid.shape[1]))
        for z in range(len(percent)):
            coloresordenados[z,] = np.array(centroid)[np.argmax(percent)]
            centroid = np.delete(centroid,np.argmax(percent),0)
            percent.pop(np.argmax(percent))

        coloresprimeros.append(rgb_to_nombre(coloresordenados[0]))

    with open(r'C:/Users/jdiez/Desktop/Base de datos.csv', 'r') as f:
        reader = csv.reader(f, delimiter = ',')
        readers = list(reader)
        contador1=0
        for i in range(len(indices)):
            lista = list(css3_db.values())
            contador=1
            if(truelabels[indices[i]] =='T-Shirt'):
                otros = ['T-Shirt','Polo','Longsleeve','Hoodie']

            elif(truelabels[indices[i]]== 'Pants'):
                otros = ['Pants','Blazer']

            elif(truelabels[indices[i]] == 'Dress'):
                otros = ['Dress', 'Skirt']

            elif(truelabels[indices[i]]== 'Shoes'):
                otros = ['Shoes']

            elif(truelabels[indices[i]]== 'Shorts'):
                otros = ['Shorts']
            while(contador<6//len(indices)):
                elegidos=[]
                for row in readers:
                    for prenda in otros:
                        if(row[1]==prenda and row[2]==coloresprimeros[i]):
                            elegidos.append(row[0])

                if(contador1+ len(elegidos)>6):
                    chosen = random.sample(elegidos, (6//len(indices))-
    contador1)
                else:
                    chosen = random.sample(elegidos, len(elegidos))

                for s in chosen:
                    paso =138
                    img=Image.open('C:/Users/jdiez/Desktop/TFM/clothing-dataset
    -master/images/'+ s.strip() + '.jpg')
                    img= img.resize((128,128))
                    test = ImageTk.PhotoImage(img)

                    label1 = tk.Label(image=test)
                    label1.image = test

                    canvas1.create_window(150 + (contador + i*(6//len(indices))
    -1)*paso, 465, window=label1)
```

```
173
174                     contador=contador+1
175                     contador1 = contador1+1
176
177                 lista.remove(css3_db[coloresprimeros[i]])
178                 color = encontrarcolor(coloresprimeros[i], lista)
179                 coloresprimeros[i] = str(rgb_to_nombre(hex_to_rgb(color)))
180
181
182
183
184 def recomendacionprenda():
185     '''
186     Funcion que recomienda ropa en funcion del tipo de prenda subida por el
        usuario. Esta funcion permite visualizar las seis recomendaciones.
187     '''
188
189     dir_img = entry1.get()
190     label7 = tk.Label(root, text= 'Las prendas recomendadas son:',font=('
        helvetica', 10), bg = '#fad7a0')
191     canvas1.create_window(185, 375, window=label7)
192     truelabels = ['Fondo','T-Shirt','Pants', 'Dress', 'Shorts', 'Shoes']
193     height =128
194     width = 128
195
196     model = tf.keras.models.load_model('modelofinal.h5')
197     Z=[]
198     imagen=getImageArr( dir_img , width , height )
199     Z.append(imagen)
200     Z=np.array(Z)
201     y_pred = model.predict(Z)
202     y_predi = np.argmax(y_pred, axis=3)
203     aux=y_predi[0].shape[0]//10
204     recort=y_predi[0][aux:y_predi[0].shape[0]-aux,aux:y_predi[0].shape[0]-aux]
205     values, counts = np.unique(recort, return_counts=True)
206     porcentajes=[]
207     for k in range(1,len(values)):
208         porcentajes.append(counts[k]/(np.sum(counts[1:len(counts)])))
209
210     aux1=np.max(porcentajes)
211     indices=[]
212     for l in range(len(porcentajes)):
213         if(aux1- porcentajes[l]<(1/len(porcentajes))):
214             indices.append(values[l+1])
215
216     with open(r'C:/Users/jdiez/Desktop/Base de datos.csv', 'r') as f:
217         reader = csv.reader(f, delimiter = ',')
218         readers = list(reader)
219         for i in range(len(indices)):
220             lista = list(css3_db.values())
221             contador=1
222             while(contador<6//len(indices)):
223                 elegidos=[]
224                 for row in readers:
225                     if(row[1]==truelabels[indices[i]]):
226                         elegidos.append(row[0])
227
228                 if(len(elegidos)>=6//len(indices)):
229                     chosen = random.sample(elegidos, 6//len(indices))
230
231                 else:
232                     chosen = random.sample(elegidos, len(elegidos))
233
```

```
234                 for s in chosen:
235                     paso =138
236                     img=Image.open('C:/Users/jdiez/Desktop/TFM/clothing-dataset
    -master/images/'+ s.strip() + '.jpg')
237                     img= img.resize((128,128))
238                     test = ImageTk.PhotoImage(img)
239 
240                     label5 = tk.Label(image=test)
241                     label5.image = test
242 
243                     canvas1.create_window(150 + (contador+ i*(6//len(indices))
    -1)*paso, 465, window=label5)
244 
245 
246                     contador=contador+1
247 
248     button3['text']='Recomiendame mas'
249 
250 
251 def clear():
252     '''
253     Funcion que borra el contenido del panel donde se ha escrito el directorio.
254     '''
255     entry1.delete(0, tk.END)
256 
257 
258 button1 = tk.Button(root, text='Cargar imagen', command=cargaimagen, bg='gray',
     fg='white', font=('helvetica', 9, 'bold')) #Boton que implementa la funcion
     de carga de imagen.
259 canvas1.create_window(770, 140, window=button1)
260 
261 button2 = tk.Button(root, text='Recomendacion en funcion de los colores',
    command=recomendacion, bg='brown', fg='white', font=('helvetica', 9, 'bold')
    ) #Boton que implementa la funcion de recomendacion basada en tipo de prenda
     y colores.
262 canvas1.create_window(355, 340, window=button2)
263 
264 button3 = tk.Button(root, text='Recomendacion en funcion de la prenda', command
    =recomendacionprenda, bg='blue', fg='white', font=('helvetica', 9, 'bold'))
    #Boton que implementa la funcion de recomendacion basada en el tipo de
    prenda.
265 canvas1.create_window(605, 340, window=button3)
266 
267 button4 = tk.Button(root, text='Borrar', command=clear, bg='gray', fg='white',
    font=('helvetica', 9, 'bold')) #Boton que implementa la funcion de borrado.
268 canvas1.create_window(235, 140, window=button4)
269 
270 root.mainloop()
```

Bibliografía

[1] I. ALDEA-BLASCO, *Redes Neuronales Convolucionales. Aspectos teóricos y aplicaciones en aprendizaje supervisado*, Universidad de Zaragoza, 2019, https://zaguan.unizar.es/record/86657?ln=es.

[2] M. CHARRAD, N. GHAZZALI, V. BOITEAU, A. NIKNAFS, *NbClust: An R package for Determining the Relevant Number of Clusters in a Data Set*, Journal of Statistical Software, vol.61, pp. 1-36, 2014, https://www.jstatsoft.org/article/view/v061i06.

[3] A. CONNEAU, H. SCHWENK, Y. LECUN, *Very Deep Convolutional Network for Text Classification*, arXiv, 2016, https://arxiv.org/abs/1606.01781.

[4] DATASET1, https://drive.google.com/file/d/0B0d9ZiqAgFkiOHR1NTJhWVJMNEU/view?resourcekey=0-3WcVUCySTS9ajn1UWHrmtA.

[5] N. DHANACHANDRA, K. MANGLEM, Y. JINA-CHANU, *Image Segmentation using K-means Clustering Algorithm and Subtractive Clustering Algorithm*, Procedia Computer Science, vol. 54, pp. 764-771, 2015.

[6] V. DUMOULIN, F. VISIN, *A guide to convolution arithmetic for deeplearning*, arXiv, 2018, https://arxiv.org/pdf/1603.07285.pdf.

[7] M. EHTASHAM-BILLAH, *Classifying Microscopic Images for Acute Lymphoblastic Leukemia using Bayesian Convolutional Neural Networks*, Örebro University, 2018.

[8] C. ENYINNA-NWANKPA, W. IJOMAH, A. GACHAGAN, S. MARSHALL, *Activation Functions: Comparison of Trends in Practice and Research for Deep Learning*, arXiv, 2018, https://arxiv.org/pdf/1811.03378.pdf.

[9] A. GARCIA-GARCIA, S. ORTS-ESCOLANO, S. OPREA, V. VILLENA-MARTINEZ, J. GARCIA-RODRIGUEZ, *A Review on Deep Learning Techniques Applied to Semantic Segmentation*, arXiv, 2017, http://arxiv.org/abs/1704.06857.

[10] F. GIMÉNEZ-PALOMARES, J. A. MONSORIU, E. ALEMANY-MARTÍNEZ, *Aplicación de la convolución de matrices al filtrado de imágenes*, Modelling in Science Education and Learning, vol. 9, pp. 97-108, 2016.

[11] GIMP, http://www.gimp.org.es.

[12] I. GOODFELLOW, Y. BENGIO, A. COURVILLE, *Deep Learning*, MIT Press, 2016, http://www.deeplearningbook.org.

[13] A. GRIGOREV, *Clothing dataset*, Medium, 2020, https://medium.com/data-science-insider/clothing-dataset-5b72cd7c3f1f.

[14] *HTML Color Names*, https://www.w3schools.com/colors/colors_names.asp.

[15] JUPYTER NOTEBOOK, *The Jupyter Notebook*, https://jupyter.org/.

[16] KERAS, *Keras: The Python Deep Learning Library*, https://keras.io/.

[17] D. KINGMA, J. LEI-BA, *ADAM: A Method for stochastic optimization*, International Conference on Learning Representations, San Diego, 2015, https://arxiv.org/pdf/1412.6980.

[18] F. KRÜGER, *Activity, Context, and Plan Recognition with Computational Causal Behaviour Models*, PhD Thesis University of Rostock, 2016, https://www.researchgate.net/publication/314116591_Activity_Context_and_Plan_Recognition_with_Computational_Causal_Behaviour_Models.

[19] LABELBOX, *Image segmentation made fast and intuitive*, https://labelbox.com/product/image-segmentation.

[20] Y. LECUN, Y. BENGIO, G. HINTON, *Deep Learning*, Nature, vol. 521, pp. 436-444, 2015, https://www.nature.com/articles/nature14539.pdf.

[21] J. MA, *Segmentation Loss Odyssey*, arXiv, 2020, https://arxiv.org/pdf/2005.13449.pdf.

[22] S. MANEEWONGVATANA, D. MOUNT, *Analysis of Approximate Nearest Neighbor Searching with Clustered Point Sets*, arXiv, 1999, https://arxiv.org/abs/cs/9901013.

[23] H. NOH, S. HONG, B. HAN, *Learning Deconvolution Network for Semantic Segmentation*, arXiv, 2015, https://arxiv.org/pdf/1505.04366.pdf.

[24] F. RICCI, L. ROKACH, B. SHAPIRA, P. KANTOR, *Recommender Systems Handbook*, Springer, 2010, https://www.cse.iitk.ac.in/users/nsrivast/HCC/Recommender_systems_handbook.pdf.

[25] S. RUDER, *An overview of gradient descent optimization algorithms*, arXiv, 2017, https://arxiv.org/abs/1609.04747v2.

[26] N. SARDANA, *Fully Convolutional Networks*, 2017, https://user.tjhsst.edu/2018nsardana/fcn.pdf.

[27] SCIKIT-LEARN, *K-means*, https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html.

[28] F. SULTANA, A. SUFIAN, P. DUTTA, *Evolution of Image Segmentation using Deep Convolutional Neural Network: A Survey*, Knowledge-Based Systems, vol. 201-202, pp. 106062, 2020.

[29] TENSORFLOW, *An end-to-end open source machine learning platform*, https://www.tensorflow.org/.

[30] K. THIRUVENKADAM, S. PADMANABAN, V. RAVINDRAN, *A Study on Validation Metrics of Digital Image Processing*, Computational Methods, Communication Techniques and Informatics conference, 2017, https://www.researchgate.net/publication/313764533_A_Study_on_Validation_Metrics_of_Digital_Image_Processing.

[31] TKINTER, *Python interface to Tcl/Tk*, https://docs.python.org/3/library/tkinter.html.

[32] WALLAPOP, https://es.wallapop.com/.

[33] S. ZHANG, L. YAO, A. SUN, Y. TAI, *Deep Learning based Recommender System: A Survey and New Perspectives*, arXiv, 2017, https://arxiv.org/abs/1707.07435.